VMS Device Support Manual

Order Number: AA–PBPWA–TE

June 1990

This manual describes how to write a driver for a device connected to a VAX processor. It discusses the required and optional components of a driver and explains their functions. It details the requirements VMS imposes upon driver code and includes guidelines for creating, loading, and debugging a driver that can run on VMS uniprocessing and multiprocessing systems.

Revision/Update Information:	This book supersedes the *VMS Device Support Manual*, Version 5.0. The reference material that was in the original appendixes is now in the *VMS Device Support Reference Manual*, Version 5.4.
Software Version:	VMS Version 5.4

digital equipment corporation
maynard, massachusetts

June 1990

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by Digital Equipment Corporation or its affiliated companies.

Restricted Rights: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

The postpaid Reader's Comments forms at the end of this document request your critical evaluation to assist in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

CDA	DEQNA	MicroVAX	VAX RMS
DDIF	Desktop–VMS	PrintServer 40	VAXserver
DEC	DIGITAL	Q-bus	VAXstation
DECdtm	GIGI	ReGIS	VMS
DECnet	HSC	ULTRIX	VT
DECUS	LiveLink	UNIBUS	XUI
DECwindows	LN03	VAX	
DECwriter	MASSBUS	VAXcluster	digital™

The following is a third-party trademark:

PostScript is a registered trademark of Adobe Systems Incorporated.

ZK5502

Production Note

This book was produced with the VAX DOCUMENT electronic publishing system, a software tool developed and sold by Digital. In this system, writers use an ASCII text editor to create source files containing text and English-like code; this code labels the structural elements of the document, such as chapters, paragraphs, and tables. The VAX DOCUMENT software, which runs on the VMS operating system, interprets the code to format the text, generate a table of contents and index, and paginate the entire document. Writers can print the document on the terminal or line printer, or they can use Digital-supported devices, such as the LN03 laser printer and PostScript printers (PrintServer 40 or LN03R ScriptPrinter), to produce a typeset-quality copy containing integrated graphics.

Contents

Part IV BUS SPECIFICS AND ADVANCED TOPICS

FIGURES

TABLES

Preface

The *VMS Device Support Manual* provides information needed to write a device driver that runs under VMS Version 5.4 and to load it into the operating system. Digital makes no guarantee that drivers written for earlier versions of VMS will execute without modification on this version of the operating system. Although the intent is to maintain the existing interface, some unavoidable changes might occur as new features are added.

The use of internal executive interfaces other than those described in this manual is discouraged.

Intended Audience

This manual is intended for system programmers who are already familiar with VAX processors and the VMS operating system.

Document Structure

This manual contains five parts.

Part I describes the components and environment of a device driver and provides explanations of VMS concepts critical to an understanding of a device driver's functions and role in the operating system. Part I contains four chapters.

- Chapter 1 describes the role of a device driver in the VMS operating system, introduces the components of a typical driver and the data structures it uses, and provides an overview of system concepts critical to driver operation. It concludes with an examination of the I/O subsystems of VAX processing systems.
- Chapter 2 provides an example of a device driver—the VMS line printer driver. It illustrates the functions of the various components of this driver and describes the driver's interaction with VMS.
- Chapter 3 discusses VMS synchronization mechanisms: interrupt priority levels; spin locks, fork locks, and device locks; fork processes and fork queues; and resource-wait queues.
- Chapter 4 provides an overview of I/O processing and discusses the interaction of device drivers with VMS.

Part II of this document describes how to code each part of a driver. It includes seven chapters.

- Chapter 5 explains some general driver coding rules and conventions and describes a device driver.
- Chapter 6 describes how to create driver tables, including the driver prologue table, driver dispatch table, and function decision table (FDT).

- Chapter 7 explains how to write FDT routines, how to use VMS-supplied FDT routines, and how to transfer control out of I/O request preprocessing.
- Chapter 8 discusses the components of a driver's start-I/O routine.
- Chapter 9 discusses the functions performed by an interrupt service routine (ISR).
- Chapter 10 describes how to perform device-dependent I/O completion and to write timeout handling routines.
- Chapter 11 describes unit and controller initialization routines, cancel-I/O routines, error logging routines, register dumping routines, and cloned unit control block (UCB) routines.

Part III of this document describes how to load and debug a device driver. This part contains two chapters.

- Chapter 12 examines the methods by which a device is logically connected to the processor and by which a driver is loaded into the operating system.
- Chapter 13 describes the use of XDELTA as a device driver debugging tool.

Part IV contains discussions of bus-specific and processor-specific details that affect the composition and operation of a device driver. The chapters in this part also discuss advanced topics relating to the writing of specific types of drivers.

- Chapter 14 discusses I/O bus features that govern the operation of direct-memory-access (DMA) transfers and that affect the code of DMA device drivers for UNIBUS and Q22 bus devices.
- Chapter 15 describes strategies for producing a MASSBUS device driver.
- Chapter 16 describes special coding considerations for generic VAXBI devices.
- Chapter 17 provides information on creating a third-party SCSI class driver to support a non-Digital-supplied small computer system interface (SCSI) device.
- Chapter 18 discusses the components of terminal class and port drivers.
- Chapter 19 describes the connect-to-interrupt driver interface that is available to real-time users.

Part V contains driver program templates and code examples of drivers for devices connected to a SCSI bus, a UNIBUS, and a Q22 bus.

- Appendix A includes a template for a UNIBUS or Q22 bus device driver.
- Appendix B includes a template for a SCSI class driver.
- Appendix C includes a sample driver that operates an RL01/RL02-type disk on the UNIBUS or Q22 bus.

- Appendix D contains a sample driver for two connected DR11 controllers on the UNIBUS or Q22 bus.
- Appendix E describes the differences between drivers intended for a VMS uniprocessing environment and those intended for a VMS multiprocessing environment. It further describes those changes required for the upgrade of non-Digital-supplied drivers written before VMS Version 5.0 and also discusses the means by which a uniprocessing driver can be converted to a multiprocessing driver.

The Glossary defines vocabulary that pertains to device drivers and their environment.

Associated Documents

Before reading the *VMS Device Support Manual*, you should have an understanding of the material discussed in the following documents:

- The *VMS Device Support Reference Manual*, a companion document, which describes the required reference material for driver programming
- *VAX Hardware Handbook*
- I/O-related portions of the *VMS System Services Reference Manual*
- The section on VMS naming conventions in the *Guide to Creating VMS Modular Procedures*
- *VMS I/O User's Reference Manual: Part I* and *VMS I/O User's Reference Manual: Part II*

You may also find useful some of the material in your processor's hardware documentation, as well as that in the following books:

- *VMS System Dump Analyzer Utility Manual*
- *Guide to Maintaining a VMS System*
- *VAX/VMS Internals and Data Structures*
- *VMS Delta/XDelta Utility Manual*

Before reading the SCSI information in Chapters 1 and 17, you should have an understanding of the material discussed in the following documents:

- American National Standard for Information Systems—Small Computer System Interface—2 (SCSI–2) specification (X3T9.2/86–109)

  The SCSI–2 specification is a draft of a proposed standard. Until it is approved, copies of this document may be purchased from Global Engineering Documents, 1990 M Street NW, Suite 400, Washington, D. C., 20036, United States; or by calling telephone number (800) 854-7179. Please refer to document X3.131–199X.
- American National Standard for Information Systems—Small Computer System Interface specification (X3.131–1986)

Copies of this document may be obtained from the American National Standards Institute, Inc., 1430 Broadway, New York, New York, 10018. This document is now known as the SCSI–1 standard.

Digital publishes two additional documents to help third-party vendors prepare SCSI peripherals and peripheral software for use with Digital's workstations and MicroVAX systems:

- The *Small Computer System Interface: An Overview* (EK–SCSIS–OV–001) provides a general description of Digital's SCSI third-party support program.
- The *Small Computer System Interface: A Developer's Guide* (EK–SCSIS–SP–001) presents the details of Digital's implementation of SCSI within its operating systems.

Conventions

This manual describes code transfer operations in three ways:

1. The phrase "issues a system service call" implies the use of a CALL instruction.
2. The phrase "calls a routine" implies the use of a JSB or BSB instruction.
3. The phrase "transfers control to" implies the use of a BRB, BRW, or JMP instruction.

Typographical conventions used in this book include the following:

- Generally, terms that are further explained in the glossary of this manual first appear in bold print. For example:

  Under the VMS operating system, a **device driver** is a set of routines and tables that the system uses to process an I/O request for a particular device type.

- Terms that serve as arguments to macros appear in boldface in the text of the manual. For example:

  If an at sign (@) character precedes the **oper** argument, then the **exp** argument describes the address of the data with which to initialize the field.

- In examples, a key name is shown enclosed to indicate that you press a named key on the terminal. For instance:

```
driver-base-address,0;X  Return
```

- A sequence such as Ctrl/x indicates that you must hold down the key labeled Ctrl while you simultaneously press another key. For instance:

```
$ CREATE MYDRIVER.OPT
BASE=0
Ctrl/Z
```

- A horizontal ellipsis indicates that additional parameters, values, or information can be entered. For example:

```
$  LINK /NOTRACE MYDRIVER1[,MYDRIVER2,...],-
       MYDRIVER.OPT/OPTIONS,-
       SYS$SYSTEM:SYS.STB/SELECTIVE_SEARCH
```

- Brackets indicate that the enclosed item is optional. For example:

  DSBINT [ipl] [,dst]

  However, brackets are not optional in the syntax of a directory name in a file specification or in the syntax of a substring specification in an assignment statement.

- In interactive examples in printed editions of this book, all output lines or prompting characters that the system prints or displays appear in black type. User-entered commands are shown in red type. In online editions of this book, user-entered commands are shown in boldface type.

  For example:

```
>>>  DEPOSIT R3 0
>>>  @DMAXDT
SYSBOOT>
SYSBOOT>  CONTINUE
```

- A vertical ellipsis means either that not all data that the system would display in response to the particular command is shown or that not all data a user would enter is shown. For example:

```
JSB     @UCB$L_FPC(R5)          ; Restore the driver process.
   .
   .
   .
;Between these instructions, the interrupt service routine
;can make no assumptions about the contents of R0 through R4.
   .
   .
   .
POPR    #^M<R0,R1,R2,R3,R4,R5> ; Restore interrupt registers.
```

- In text, an arrow (→) between abbreviated names or objects functions as a pointer to illustrate a pointing chain between structures in the database. In the following example, the UCB points to the DDT data structure, which points to the FDT address.

  UCB → DDT → FDT address

1 Introduction to Device Drivers

Under the VMS operating system, a **device driver** is a set of routines and tables that the system uses to process an I/O request for a particular device type.

The VMS operating system's approach to I/O is that the system should perform as much of the processing of an I/O request as possible and that drivers should restrict themselves to the device-specific aspects of I/O processing. To accomplish this, the VMS operating system provides drivers with the following services:

- A Queue I/O request ($QIO) system service that preprocesses an I/O request by performing those functions and checks that are common to all devices; for example, validating those arguments of the I/O request that are not device specific
- Many operating system routines that drivers can call to perform I/O preprocessing, allocate and deallocate resources, and synchronize driver execution
- Macros that drivers can invoke to accomplish tasks that would otherwise require many lines of code
- A VMS I/O postprocessing routine that performs device-independent I/O postprocessing for all I/O requests

Thus, drivers can leave the device-independent I/O processing to the operating system and can concentrate on servicing those aspects of an I/O operation that vary from device type to device type. In addition, drivers can call VMS system routines to perform many functions that are common to several, but not all, devices.

A device driver does not run sequentially from beginning to end. Rather, the operating system uses driver tables and other information maintained by itself and the driver to determine which driver routines to activate and when they should be activated. Because little sequential processing of driver code occurs, the VMS operating system must assume the responsibility for synchronizing the execution of the various driver routines, as well as the execution of all drivers in the system. A major purpose of this book is to describe the conventions that all VMS drivers must follow to maintain this synchronization and cooperate with the operating system in I/O request processing.

This section first defines the general functions and purposes of a VMS device driver. It then introduces VMS concepts crucial to an understanding of how device drivers work within the operating system and integral to the process of successfully writing one. It concludes with a brief description of the flow of driver activity in servicing an I/O request, using the VMS line printer driver as an example.

1.1 Driver Functions

A system utility loads a VMS device driver into system virtual address space and creates its associated data structures. Once loaded, a device driver controls I/O operations on a peripheral device by performing the following functions:

- Defining the peripheral device for the rest of the operating system
- Preparing a device unit and its controller (or both) for operation at system start-up and during recovery from a power failure
- Performing device-dependent I/O preprocessing
- Translating programmed requests for I/O operations into device-specific commands
- Activating a device unit
- Responding to hardware interrupts generated by a device unit
- Responding to device timeout conditions
- Responding to requests to cancel I/O on a device unit
- Reporting device errors to an error logging program
- Returning status from a device unit to the process that requested the I/O operation

1.2 Driver Components

Normally, a device driver module can consist of the routines and tables discussed in this section. With a few exceptions, which are noted throughout Chapter 6, the order of the various routines and tables within the driver module is not important.

1.2.1 Driver Tables

The following tables appear in every driver.

The **driver prologue table** (DPT) defines the identity and size of the driver to the system utility that loads the driver into virtual memory and creates the associated data structures. With the information provided in the DPT, the driver-loading procedure can both load and reload drivers and perform the I/O database initialization that is appropriate to either situation.

Section 6.1 describes the procedure for creating a DPT and further discusses its functions. The DPT contents are shown and described in the *VMS Device Support Reference Manual.*

The **driver dispatch table** (DDT) lists the addresses of the entry points of standard routines within the driver, and records the size of the diagnostic and error message buffers for drivers that perform error logging. You can find additional information and instructions on how to specify a DDT in Section 6.2. The structure and contents of the DDT are shown and described in the *VMS Device Support Reference Manual.*

The **function decision table** (FDT) lists all valid function codes for the device, and associates valid codes with the addresses of I/O preprocessing routines, called **FDT routines**. The driver contains device-dependent FDT routines, and the VMS operating system itself provides routines (described in Section 7.5) that perform request preprocessing common to many I/O functions.

When a user process calls the $QIO system service, the system service uses the I/O function code specified in the request to traverse the FDT and select one or more of these preprocessing routines for execution, as appropriate to the function. To prepare for the actual I/O operation, FDT routines perform such tasks as allocating buffers in system space, locking pages in memory, and validating the device-dependent arguments (**p1** to **p6**) of the $QIO request. Section 6.3 provides further discussion of the FDT, and Chapter 7 details strategies and rules for writing, specifying, and exiting from an FDT routine.

1.2.2 Driver Routines

In addition to any FDT routines it may contain, a device driver generally contains both a start-I/O routine and an interrupt service routine.

The **start-I/O routine** performs such additional device-dependent tasks as translating the I/O function code into a device-specific command, storing the details of the user request in the device's unit control block (UCB) in the I/O database and, if necessary, obtaining access to controller and adapter resources. Whenever the start-I/O routine must wait for these resources to become available, the VMS operating system suspends the routine, reactivating it when the resources become free.

The start-I/O routine ultimately activates the device by suitably loading the device's registers. At this stage, the start-I/O routine invokes a VMS macro that causes its execution to be suspended until the device completes the I/O operation and posts an interrupt to the processor. The start-I/O routine remains suspended until the driver's **interrupt service routine** handles the interrupt.

When a device posts an interrupt, its driver's interrupt service routine determines whether the interrupt is expected or unexpected, and takes appropriate action. If the interrupt is expected, the interrupt service routine reactivates the driver's start-I/O routine at the point of suspension. The general course of action of driver mainline code at this time is to perform device-dependent I/O postprocessing and to transfer control to the VMS operating system for device-independent I/O postprocessing.

Details on writing a start-I/O routine appear in Chapter 8. A description of a driver interrupt service routine appears in Chapter 9.

You can also include any of the following routines in a device driver.

The **unit initialization routine** and **controller initialization routine** prepare a device or controller for operation when the VMS driver-loading procedure loads the driver into memory and when the VMS system recovers from a power failure. The amount and type of initialization needed by devices and controllers vary according to the device type and

the I/O bus to which the device or controller is attached. Section 11.1 provides additional information about device driver initialization routines.

A **timeout handling routine** retries I/O operations and performs other error handling when a device fails to complete a request in a reasonable period of time. Once every second, the VMS system timer checks all devices in the system for device timeout. When it locates a device that has timed out, because it is off line or some error has occurred, the system timer calls the driver's timeout handling routine.

Depending upon the reason for the timeout, the timeout handling routine may call a VMS error logging routine to allocate and fill an error message buffer with information about the error. In turn, the error logging routine can call a **register dumping routine** in the driver that also loads into the buffer the contents of device registers at the time of the error.

Timeout handling routines are discussed in Section 10.2. Register dumping routines and driver error handling are discussed in Section 11.3.

The VMS operating system calls a driver's **cancel-I/O routine** when a user process issues a Cancel I/O on Channel ($CANCEL) system service for the device. It may also call the routine when the device's reference count goes to zero, which occurs when all users with assigned channels to the device have deassigned them. The discussion of the cancel-I/O routine appears in Section 11.2.

1.3 The I/O Database

Because a driver and the operating system cooperate to process an I/O request, they must have a common and current source of information about the request. This is the function of the I/O database. Under the VMS operating system, the I/O database consists of the following three parts:

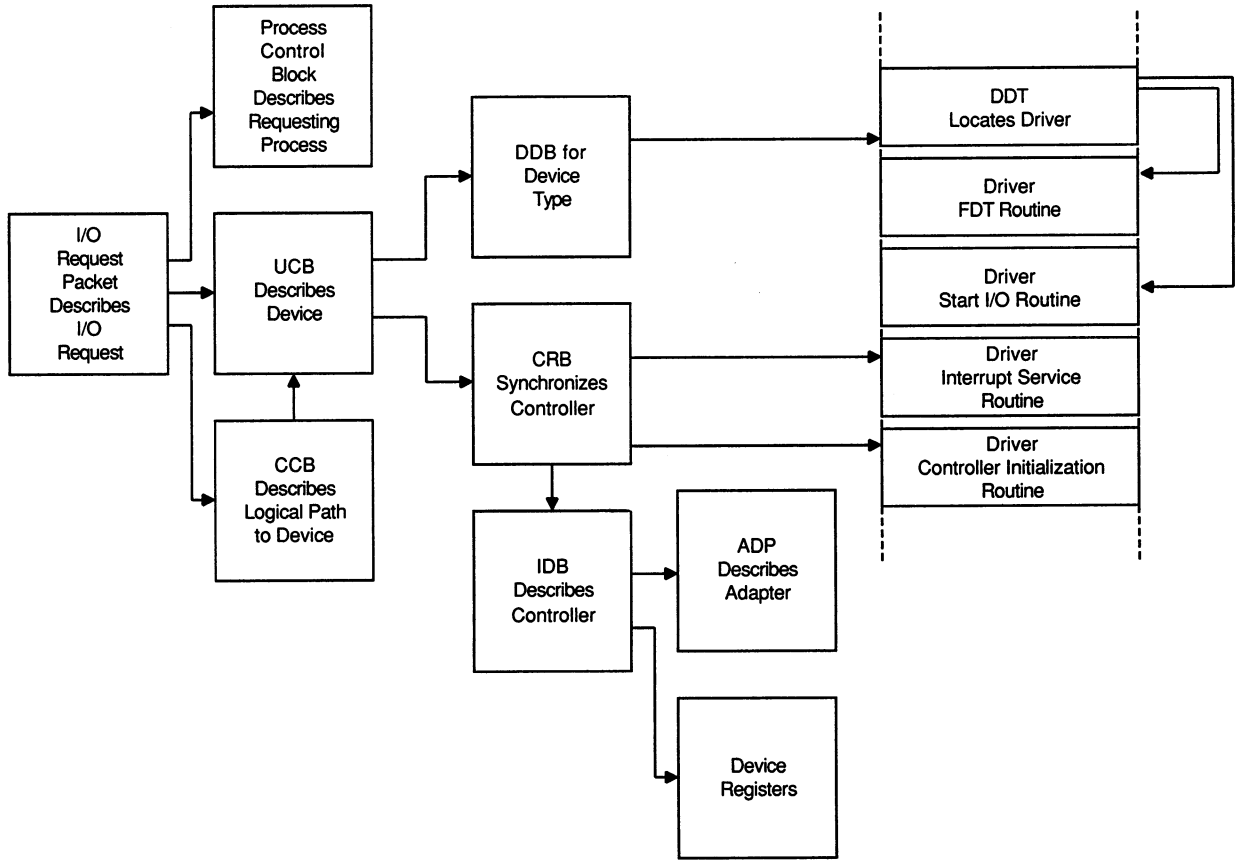
- Driver tables that allow the system to load drivers, to validate device functions, and to call driver routines at their entry points
- Data structures that describe I/O bus adapters, device types, device units, device controllers, and logical paths from processes to devices
- I/O request packets that define individual requests for I/O activity

Illustrations of I/O database structures and detailed descriptions of their fields appear in the data structure chapter of the *VMS Device Support Reference Manual*. Figure 1–1 illustrates some of the relationships among VMS I/O routines, the I/O database, and a device driver.

1.3.1 Driver Tables

The three driver tables—driver prologue table, driver dispatch table, and function decision table—are defined in every driver. Section 1.2 lists these tables among the other components of a device driver, and Chapter 6 discusses their contents.

Figure 1–1 The I/O Database



1.3.2 Data Structures

I/O database data structures describe peripheral hardware and are used by the operating system to synchronize access to devices. VMS creates these data structures either at system startup or when a driver is loaded into the system.

The system defines a **unit control block** (UCB) for each device unit attached to the system. A UCB defines the characteristics and current state of an individual device unit.

UCBs are the focal point of the I/O database. When a driver is suspended or interrupted, the UCB keeps the context of the driver in a set of fields collectively known as a **fork block.**[1] In addition, the UCB contains the listhead for the queue of pending I/O request packets (IRPs) for the unit.

[1] Other structures, such as the CRB, also include a fork block. The discussion of fork blocks and fork processes in Section 1.5 explains the role of fork blocks in driver processing.

A **device data block** (DDB) contains information common to all devices of the same type that are connected to a particular controller. It records the generic device name concatenated with the controller designator (for example, LPA, DBB), and the name and location of the associated device driver. In addition, the DDB contains a pointer to the first UCB for the device units attached to the controller.

The operating system creates a **channel request block** (CRB) for each controller. A CRB defines the current state of the controller and lists the devices waiting for the controller's data channel. It also contains the code that dispatches a device interrupt to the interrupt service routine (ISR) for that unit's driver.

The system also creates for each controller an **interrupt dispatch block** (IDB). An IDB lists the device units associated with a controller and points to the UCB of the device unit that the controller is currently servicing. In addition, an IDB points to device registers and the controller's I/O adapter.

An **adapter control block** (ADP) defines the characteristics and current state of an I/O adapter, such as the VAX UNIBUS and MASSBUS adapters, the Q22 bus interface of MicroVAX systems, or a device attached to the VAXBI bus. An ADP contains the queues and allocation bitmaps necessary to allocate the adapter's resources. VMS provides routines that drivers can call to interface with the appropriate adapter.

The **channel control block** (CCB) describes the logical path between a process and the UCB of a specific device unit.[2] Each process owns a number of CCBs. When a process issues the Assign I/O Channel ($ASSIGN) system service, the system writes a description of the assigned device to the CCB.

Unlike the data structures mentioned earlier, a CCB is not located in nonpaged system space, but in the process's control region (P1 space).

1.3.3 I/O Request Packets

The third part of the I/O database is a set of I/O request packets. When a process requests I/O activity, the operating system constructs an **I/O request packet** (IRP), that describes the I/O request in a standard form.

The IRP contains fields into which the system and driver I/O preprocessing routines can write information: for instance, the device-dependent arguments specified in the call to the $QIO system service. The packet also includes buffer addresses, a pointer to the target device's UCB, an I/O function code, and pointers to the I/O database. After preprocessing, the IRP can be queued to a list originating in the device's UCB to await processing by the driver.

[2] Channel request blocks (CRBs) and channel control blocks (CCBs) are two separate data structures. To help distinguish the two, it may be helpful to think of the channel request block as the "controller request" block because it describes the hardware controller. In contrast, the channel control block is used by a process and a device unit to manage the logical channel (the **channel** argument to the $ASSIGN and $QIO system services) in accomplishing I/O operations.

When the device unit is free and the IRP is next in line to be processed on the unit, the system sends it to the device driver's start-I/O routine. The start-I/O routine uses the IRP as its source of detailed instructions about the operation to be performed.

1.4 Synchronization of Driver Activity

Device drivers and other kernel-mode code must maintain synchronization with other priority operating system activities. The term **synchronization** refers to the means by which such code accesses shared data in a consistent, orderly, and predictable fashion. Because there may be more than one processor active in a VMS system, system-level code must synchronize its actions with other code threads it may have preempted on the same (or *local*) processor, as well as with those that are active (or to be activated) on other processors in the system. The VMS operating system uses hardware and software interrupt priority levels (IPLs) to order system events on each local processor in a VAX system. The VAX hardware defines 32 interrupt priority levels (IPLs). The higher numbered IPLs (16 to 31) are reserved for hardware interrupts, such as those posted by devices. The VMS operating system uses the lower numbered IPLs (0 to 15). Code that executes at a higher IPL takes precedence over code that executes at a lower IPL.

A driver, in concert with the operating system, ensures that it maintains system synchronization by performing certain activities and by accessing certain data only at the appropriate IPL. In a VMS multiprocessing system, the driver extends the synchronization it achieves by executing locally at a given IPL by acquiring ownership of the spin lock associated with the operation it is performing. (IPL, spin locks, and other forms of synchronization in a VMS system are discussed fully in Chapter 3.)

1.5 Driver Context

As indicated in Section 1.2.2, a driver may have several routines to which the VMS operating system may pass control in certain situations. The context in which any one routine receives control from VMS may differ substantially from that in which another receives control. It is essential that a driver routine not attempt to exceed the limitations of the context in which it executes.

In general, context is characterized by the following factors:

- The current IPL of the executing processor
- The IPL at which the thread of execution that resulted in the call to the driver began
- The currently owned spin locks of the executing processor
- The data structures available to the routine
- Data available to the routine in registers, in data structure fields, and on the stack

- The condition of the registers, data structure fields, and stack when the routine exits
- The ability or inability to access process space

A complete description of the context of each driver routine appears in the entry points chapter of the *VMS Device Support Reference Manual*. The following are some general observations:

- All device driver routines execute in kernel mode at an elevated IPL.
- Only driver FDT routines execute within process context and can access process space (P0 and P1).
- The majority of driver routines execute in *interrupt* (or *system* context): that is, in the sequence of execution that follows a processor's grant of an interrupt request at a given IPL. Such code can refer only to system (S0) space. Moreover, it cannot incur exceptions, including page faults, without causing a fatal bugcheck. Code executing in interrupt context is serviced on the interrupt stack, and must synchronize its execution with other priority code threads by using IPLs, spin locks, and resource wait queues, all of which are described in Chapter 3.

Most driver processing of an I/O request (before and after the device acknowledges the servicing of the request by requesting an interrupt from the processor) occurs at a *fork IPL*. This portion of driver code, which includes most of the start-I/O routine, is commonly known as the driver's **fork process**.

There are several instances in the processing of an I/O request when a driver fork process must suspend execution to wait for a resource or a device interrupt. To make the matter of saving and restoring fork process context as efficient as possible, the VMS operating system places a restriction on the context of a driver fork process, in addition to those that apply to any process in interrupt context. **Fork context** consists of the following:

- Two general purpose registers (R3 and R4)
- The program counter (PC)
- A fork block (usually the unit control block, the address of which is presumed to be in R5 at the time of the suspension) that can contain additional fork process context

VMS places the fork block of a suspended fork process in either a processor-specific fork queue or a resource wait queue where it waits to be resumed. When it resumes the fork process, VMS ensures that the fork context is restored. Fork blocks, fork processes, and fork queues are discussed fully in Section 3.3.3.

1.5.1 Example of Driver Context-Switching

Because a device driver consists of a number of routines that are activated by VMS, the operating system for the most part determines the context in which the routines execute.

As an example, consider the following write request that occurs without error:

1. A user process executing in user mode calls the $QIO system service to write data to a device.
2. The $QIO system service gains control in process context but in kernel mode. It performs device-independent preprocessing of the I/O request.
3. The system service uses the driver's function decision table (FDT) to call the appropriate FDT routines to perform device-dependent preprocessing. These FDT routines execute in full process context in kernel mode.
4. When preprocessing is complete, a VMS routine creates a fork process to execute the driver's start-I/O routine in kernel mode.
5. The start-I/O routine activates the device unit and suspends itself. At this point, VMS suspends the fork process executing the start-I/O routine and saves sufficient context to reactivate the start-I/O routine at the point of suspension.
6. When the device completes the data transfer, it requests an interrupt. The interrupt causes the system to activate the driver's interrupt service routine.
7. The interrupt service routine executes to handle the device interrupt. It then causes the start-I/O routine to resume in interrupt context.
8. The start-I/O routine regains control in interrupt context but almost immediately issues a request to the operating system to transform its context to that of a fork process. This action dismisses the interrupt.
9. When reactivated in fork process context, the start-I/O routine performs device-specific I/O completion and passes control to the system for additional I/O postprocessing.
10. VMS I/O postprocessing runs in interrupt context at a lower IPL and issues a special kernel-mode asynchronous system trap (AST) for the user process requesting I/O.
11. When the special kernel-mode AST is delivered, the AST routine executes in full process context in kernel mode to deliver data and status to the process. If the original request specified a user-mode AST, the special kernel-mode AST queues it.
12. When the user process gains control, the user's AST routine executes in full process context in user mode.

1.6 Hardware Considerations

The VMS operating system runs on any of the following VAX systems. It can also support non-Digital-supplied devices.

- VAX–11/780 and VAX–11/785
- VAX–11/730 and VAX–11/750
- VAX 8600/8650
- VAX 82*x*0/83*x*0
- VAX 85*x*0 and VAX 8700/88*x*0
- VAX 6000 series
- VAX 9000 series
- MicroVAX 3400/3600/3900 series
- MicroVAX/VAXstation II and VAX 4000 series
- MicroVAX/VAXstation 3100
- VAXstation 3520/3540

Although these system configurations employ the same operating system and conform to the VAX architecture, there are some differences in design among the machines that merit consideration in device driver coding, installation, and debugging. For instance, VAX systems differ in the amount of available physical address space and in the location of device registers. Some VAX systems are available in multiprocessor configurations. Also, VAX systems support different and various combinations of I/O buses to which a nonstandard device can be connected.

If you follow the conventions described in this manual when writing your driver, your driver should, with little modification, drive the same device attached to a corresponding I/O bus of another VAX system. For specific system design and device configuration information, refer to your system's technical reference or hardware manual or the *VAX Hardware Handbook*.

1.6.1 Driver Dependency on VAX Processing Systems

This section outlines some of the general differences among the VAX processing systems that have a bearing upon the development of driver code. The main thrust of the discussion is to provide a brief summary of the layout of the I/O subsystems of the VAX processing systems, to define a general terminology, and, when necessary, to direct device driver writers to documentation particular to the I/O configuration of their device.

1.6.1.1 VAX–11/780, VAX–11/785, and VAX 8600/8650 Systems

The VAX–11/780, VAX–11/785, VAX 8600 and VAX 8650 systems, from the viewpoint of I/O architecture, are SBI-based systems. That is, the **synchronous backplane interconnect** (SBI) is the bus by which I/O adapters communicate with main memory and the central processor (see Figure 1–2). I/O adapters supported by the SBI include the UNIBUS adapter (UBA), the MASSBUS adapter (MBA), and the DR780 interface adapter. Correspondingly, peripheral devices attach to either the UNIBUS, MASSBUS, or DR32 device interconnect (DDI) of the DR780 adapter. Main memory shares the SBI with the I/O adapters on the VAX–11/780 and VAX–11/785. The VAX 8600 and VAX 8650 employ a separate bus to which main memory is attached and can each be configured with up to two SBIs for I/O adapters.

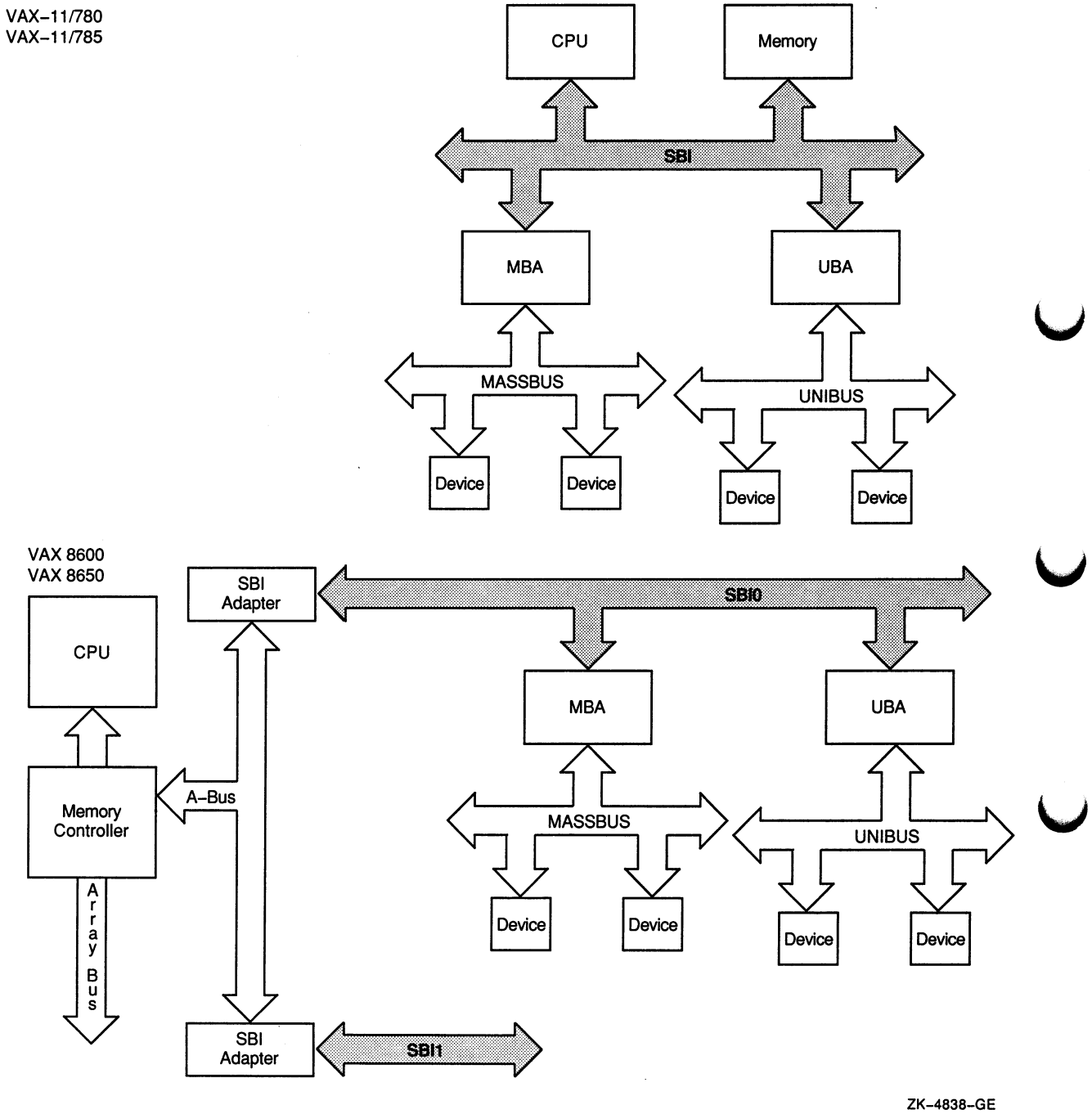
For these systems, nonstandard devices are commonly attached to the UNIBUS, although some nonstandard devices connect to the MASSBUS and DDI. The components of UNIBUS and MASSBUS drivers are nearly identical and the strategies for producing driver code are similar; writers of either type of driver will profit from reading the bulk of this manual. Writers of UNIBUS drivers can find specific information about the UNIBUS adapter and VMS support for UNIBUS drivers in Chapter 14. MASSBUS driver writers should refer to Chapter 15 for similar information about the MASSBUS. Digital supplies a device driver and an application library for DDI devices; the *VMS I/O User's Reference Manual: Part II* discusses the DR32 interface driver in detail.

A final note on terminology regarding these systems is pertinent. For the purposes of the discussion in this book, the term *VAX–11/780* refers to the family of VAX systems that includes the VAX–11/780 and VAX–11/785; the term *VAX 8600* refers to the VAX 8600 and VAX 8650.

1.6.1.2 The VAX–11/750 System

The VAX–11/750 system resembles the VAX–11/780-type systems in that it supports UNIBUS, MASSBUS, and DDI peripheral devices (see Figure 1–2). The backplane, or CPU-to-memory interconnect (CMI), by which I/O adapters communicate with the central processor and main memory, is integral to the processor, as are the UNIBUS interface (UBI) and MASSBUS adapter (MBA). The DR750 interface adapter connects the CMI to the DDI subsystem. Peripheral devices connect to the UNIBUS, MASSBUS, and DDI. A separate memory interconnect provides an interface between main memory and the rest of the system.

Figure 1–2 SBI-Based System Configurations



For the VAX–11/750, nonstandard devices are commonly connected to the UNIBUS, although some nonstandard devices attach to the MASSBUS. The components of UNIBUS and MASSBUS drivers are identical, and the strategies for developing driver code are similar. Writers of either type of driver will profit from reading this manual. Writers of UNIBUS drivers can find specific information about the UNIBUS adapter and VMS support for UNIBUS drivers in Chapter 14. MASSBUS driver writers should refer to Chapter 15 for similar information about the MASSBUS. Digital supplies a device driver and an application library for DDI devices device; the *VMS I/O User's Reference Manual: Part II* discusses the DR32 interface driver in detail.

1.6.1.3 The VAX–11/730 System

The VAX–11/730 system, like the VAX–11/750 system, incorporates an integral UNIBUS adapter to control transactions between UNIBUS peripheral devices, the processor, and the main memory interface. The VAX–11/730 does not, however, support MASSBUS devices. Writers of UNIBUS drivers can find specific information about the UNIBUS adapter and VMS support for UNIBUS drivers in Chapter 14.

1.6.1.4 VAX 82*x*0/83*x*0, VAX 85*x*0/8700/88*x*0, and VAX 6000-Series Systems

The VAX 82*x*0/83*x*0, VAX 85*x*0/8700/88*x*0, and VAX 6000 series are VAXBI-based systems; that is, the VAXBI is the bus by which I/O adapters communicate with main memory and the central processor (see Figure 1–3).

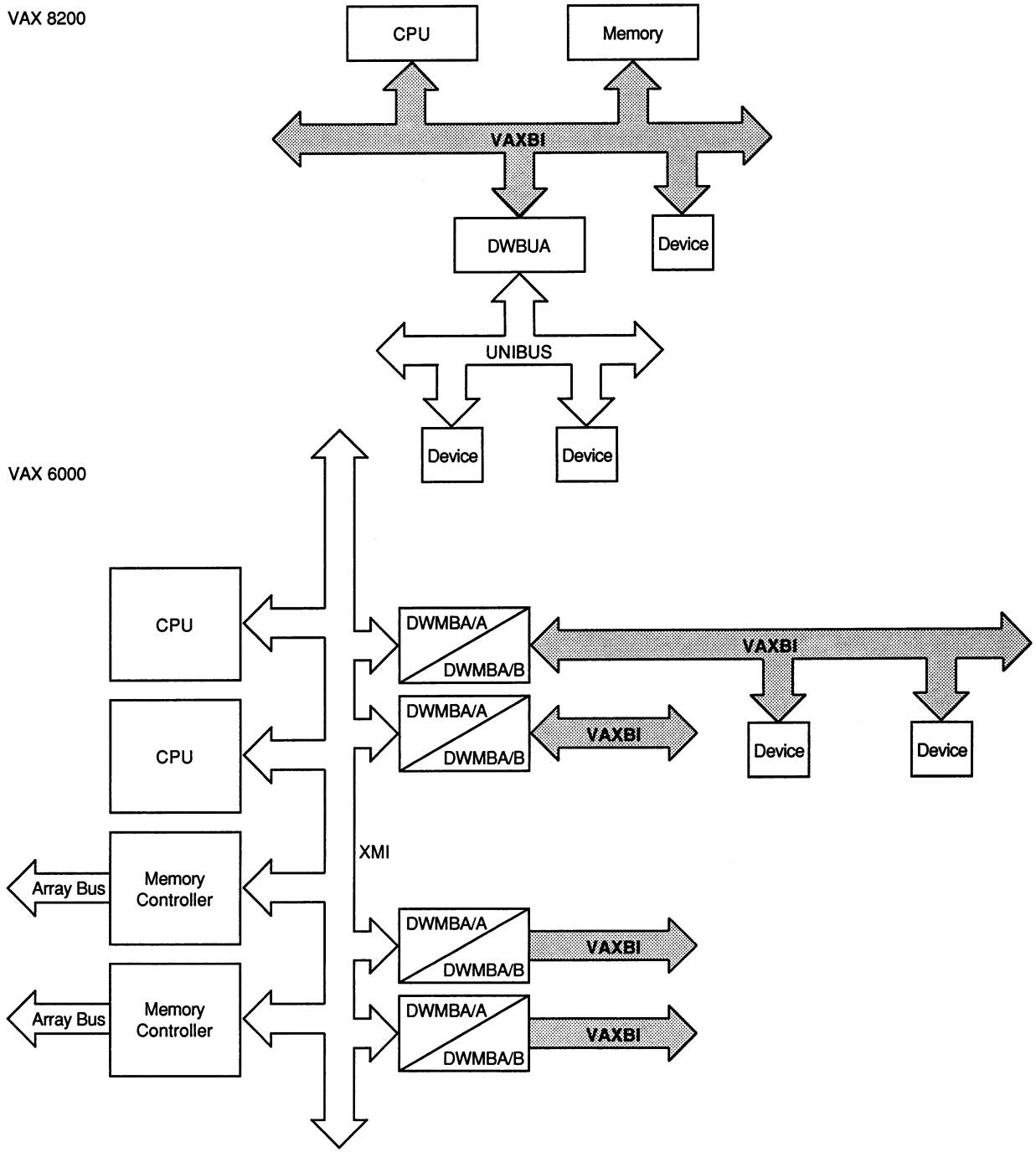
In a VAX 82*x*0/83*x*0 configuration, main memory, the DWBUA, and other devices are all connected directly to the VAXBI bus. By contrast, the VAX 85*x*0/8700/88*x*0 and VAX 6000-series configurations employ separate memory interconnects (known as the NMI, PBI, or XMI), as illustrated in Figure 1–3, to service main memory. The VAX 85*x*0/8700 /88*x*0 provides multiple VAXBI buses to which I/O adapters and devices can be attached. The VAX 83*x*0, VAX 8800/8820/8830, and VAX 6000 series are multiprocessor systems.

The VAXBI bus supports UNIBUS peripherals by means of the VAXBI-to-UNIBUS adapter (DWBUA or DWMUA). Writers of UNIBUS drivers can find specific information about the UNIBUS adapter and VMS support for UNIBUS drivers in Chapter 14.

The VAXBI also supports non-Digital-supplied devices designed according to specifications established by Digital and a license granted by Digital. Writers of drivers for such devices, referred to as **generic VAXBI devices** in this manual, can find specific information in Chapter 16.

For the purposes of the discussion in this book, the term **UNIBUS adapter** includes the DWBUA and DWMUA, and the term **backplane interconnect** represents the VAXBI bus.

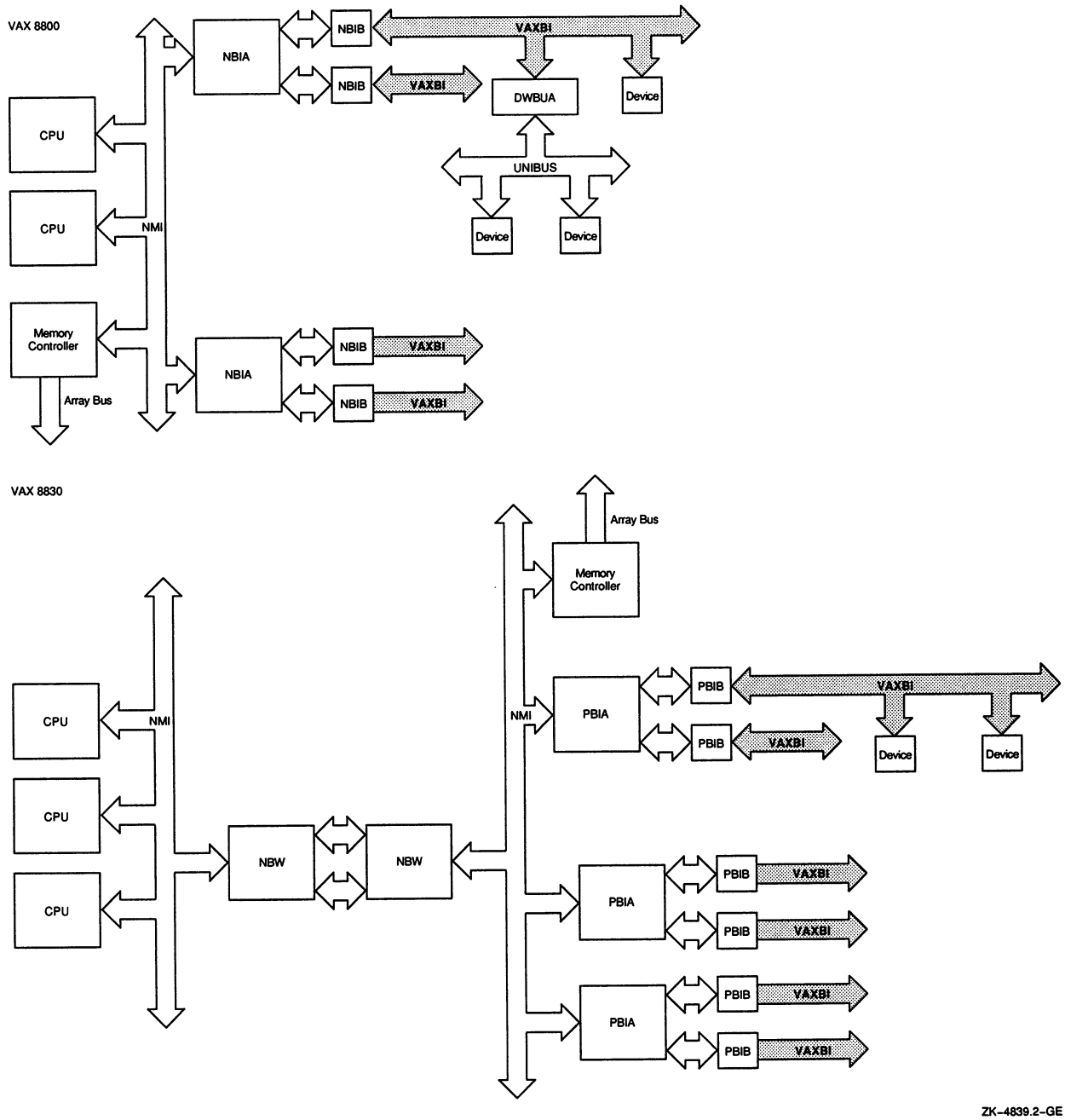
Figure 1–3 VAXBI-Based System Configurations



ZK–4839.1–GE

(continued on next page)

Figure 1–3 (Cont.) VAXBI-Based System Configurations



1.6.1.5 The VAX 9000 Series System

The VAX 9000 Series system employs the SCU/XMI bus architecture illustrated in Figure 1–4. It features a two-level I/O subsystem supporting up to four XMI buses.

A **system control unit (SCU)** and an I/O control unit translate each address and connect a VAX 9000 CPU or memory bus to a target XMI and device or bus adapter. The primary I/O bus is the XMI. It is reserved to devices and adapters supplied by Digital.

The SCU and I/O control unit connect to each XMI through an XJA adapter. Each XMI-to-VAXBI adapter (DWMBA/A) provides connection to the second level I/O subsystem. At the second level, device support is provided for non-Digital-supplied devices connected to the VAXBI bus. Writers of drivers for such devices, referred to as **generic VAXBI devices** in this manual, can find specific information in Chapter 16.

1.6.1.6 The MicroVAX 3400/3600/3900 Series, MicroVAX/VAXstation II, and VAX 4000 Series Systems

The MicroVAX 3400/3600/3900 series, the MicroVAX/VAXstation II, and the VAX 4000 series are Q22 bus-based systems. On these systems, the Q22 bus is the bus by which peripheral devices communicate with main memory and the processor.[3] Q22 bus device drivers are similar enough to those that drive UNIBUS devices that most of the discussion of UNIBUS drivers in this book can equally pertain to the writing of Q22 bus device drivers (see Chapter 14 for a discussion of the similarities and differences).

As you can see in Figure 1–5, in these systems main memory and I/O devices reside on separate interconnects. MicroVAX systems implement a scatter-gather map containing 8192 map registers that allows devices to perform multiple-block, direct-memory-access (DMA) transfers.[4]

For the purposes of discussion in this manual, the term **backplane interconnect** represents the Q22 bus in the MicroVAX systems. The term **Q22 bus interface** represents those functions performed by these processors that resemble those performed by the UNIBUS adapter of other VAX systems. In most instances, you can assume that discussions of the UNIBUS adapter apply as well to the Q22 bus.

[3] DMA controllers attached to the Q22 bus must be capable of 22-bit addressing.

[4] In these systems, the 4Mb of Q22 bus memory is located from physical address 30000000_{16} to $303F0000_{16}$. If you must install controllers that contain local memory on the Q22 bus, it is best to install them in the last 3.75Mb of Q22 bus memory (after physical address 30040000_{16}). The first 0.25Mb (256Kb) of Q22 bus memory contains 496 map registers, 127 of which must be free for use by VMS in booting. If you must place a controller containing memory in this address region, it cannot occupy more than 369 pages. If the controller exceeds this space, VMS will probably boot but will not be able to take crash dumps.

Figure 1–4 SCU/XMI-Based Systems Architecture

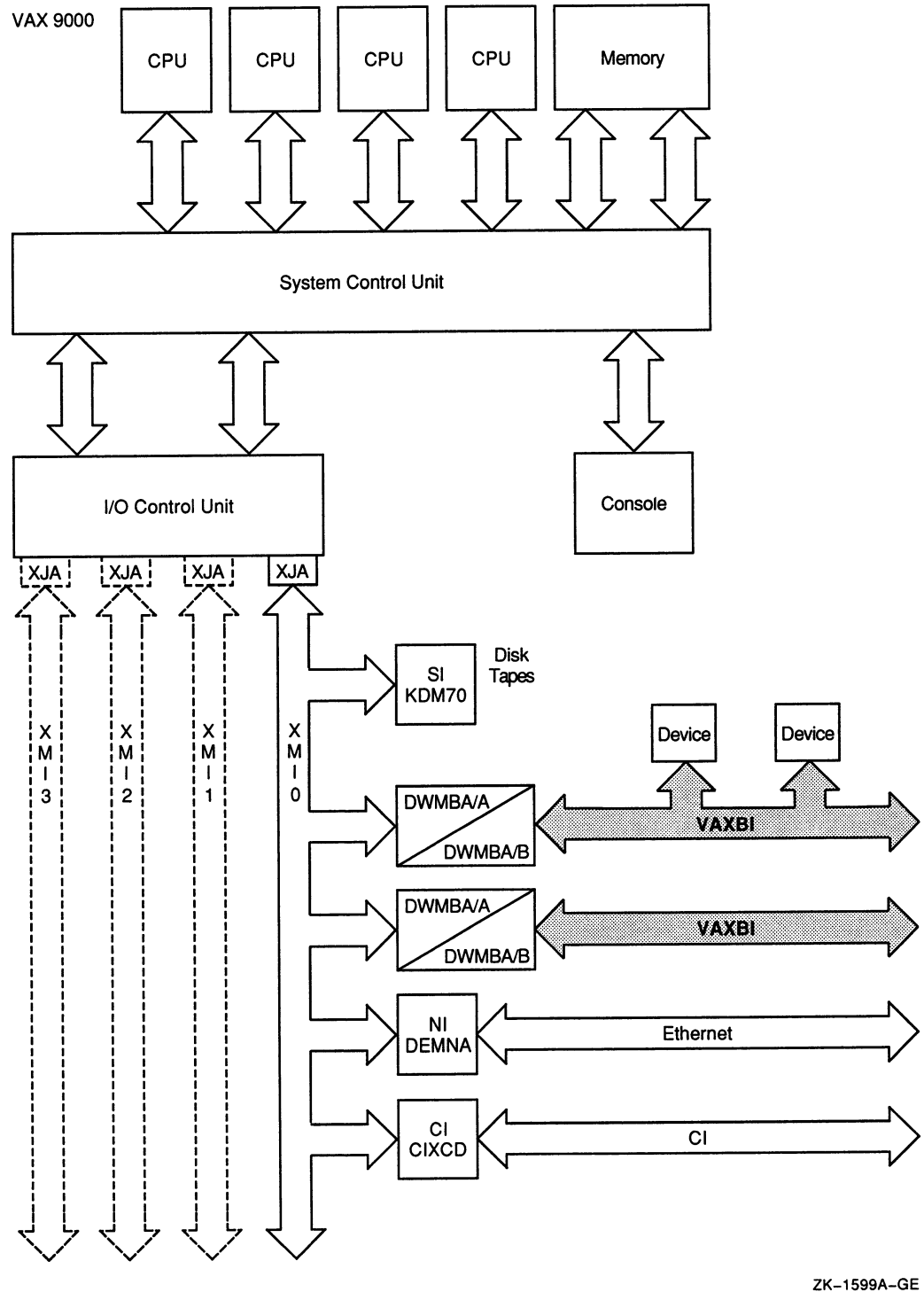
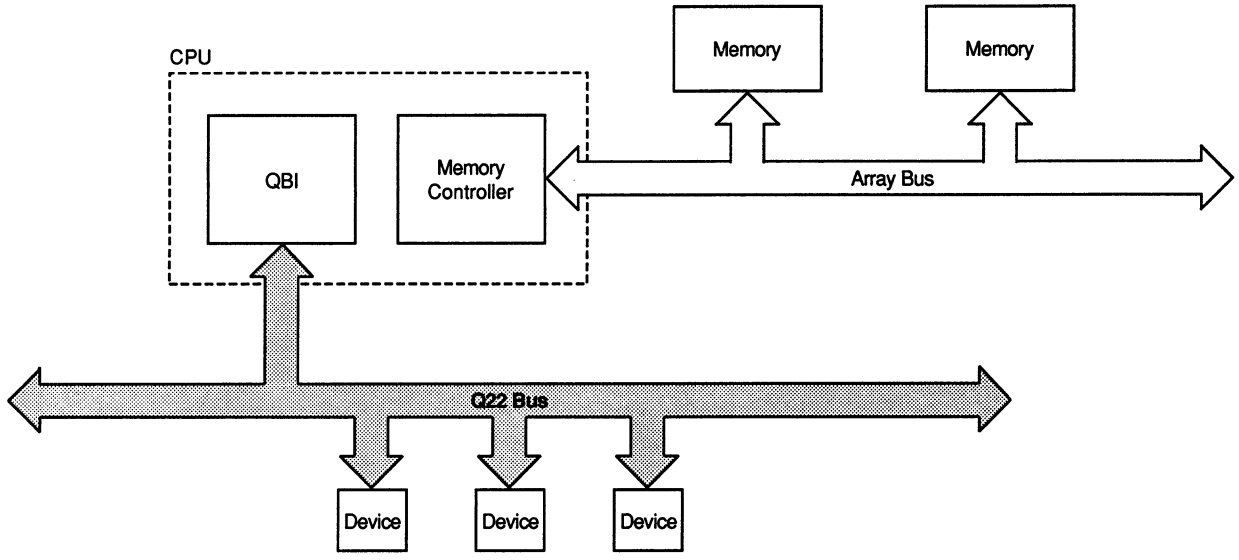


Figure 1–5 Q22 Bus Based Systems

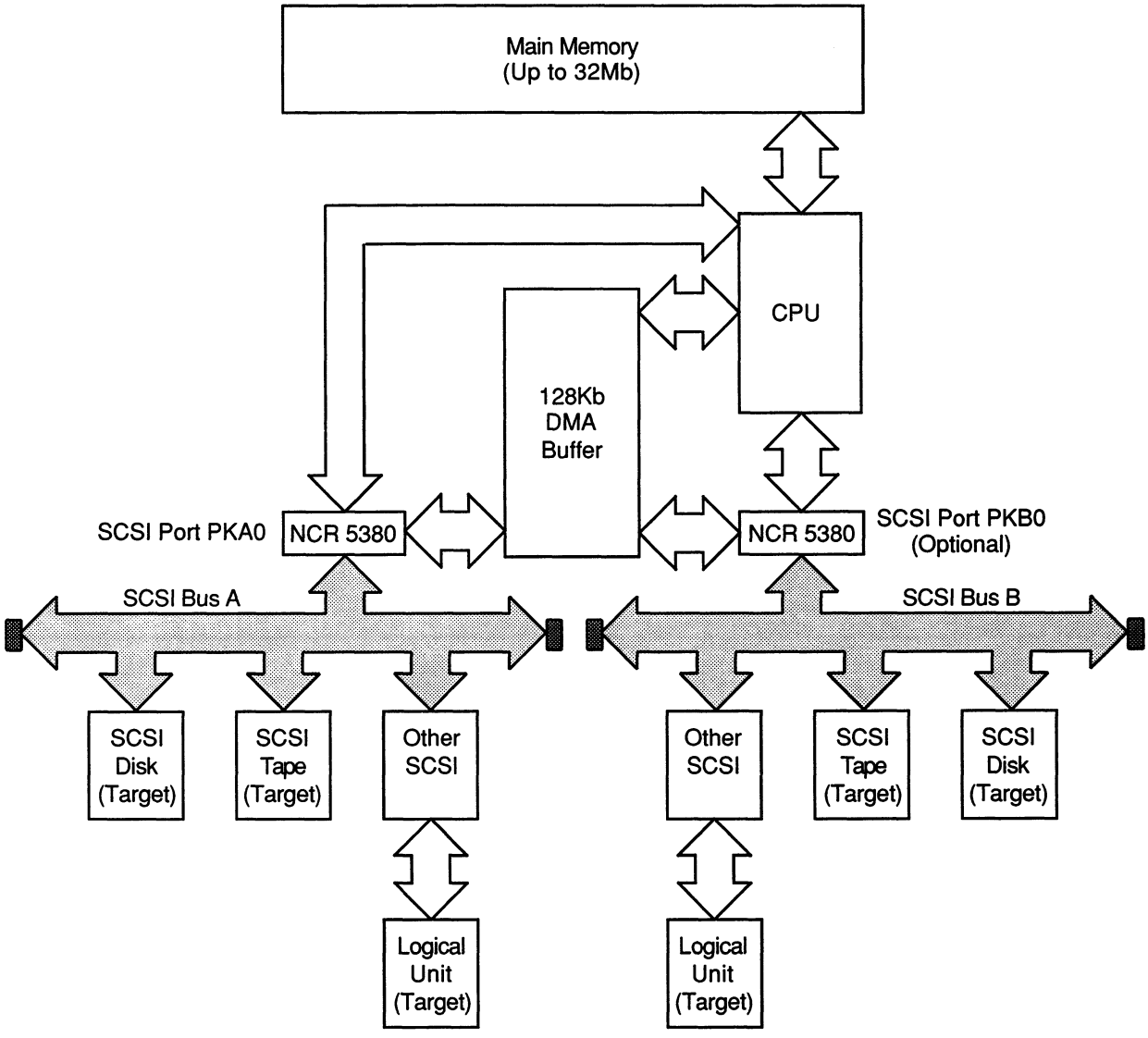


1.6.1.7 The MicroVAX/VAXstation 3100 and VAXstation 3520/3540 Systems

The VMS operating system offers a native mode implementation of the ANSI Small Computer System Interface (SCSI) bus on its MicroVAX/VAXstation 3100 and VAXstation 3520/3540 system configurations.

MicroVAX/VAXstation 3100 systems are uniprocessing systems, providing access to one or two SCSI buses, each under the control of an NCR 5380 SCSI controller chip that supports asynchronous data transfers. MicroVAX/VAXstation 3100 systems support the SCSI asynchronous event notification feature. Figure 1–6 shows a representative configuration of a MicroVAX/VAXstation 3100 system.

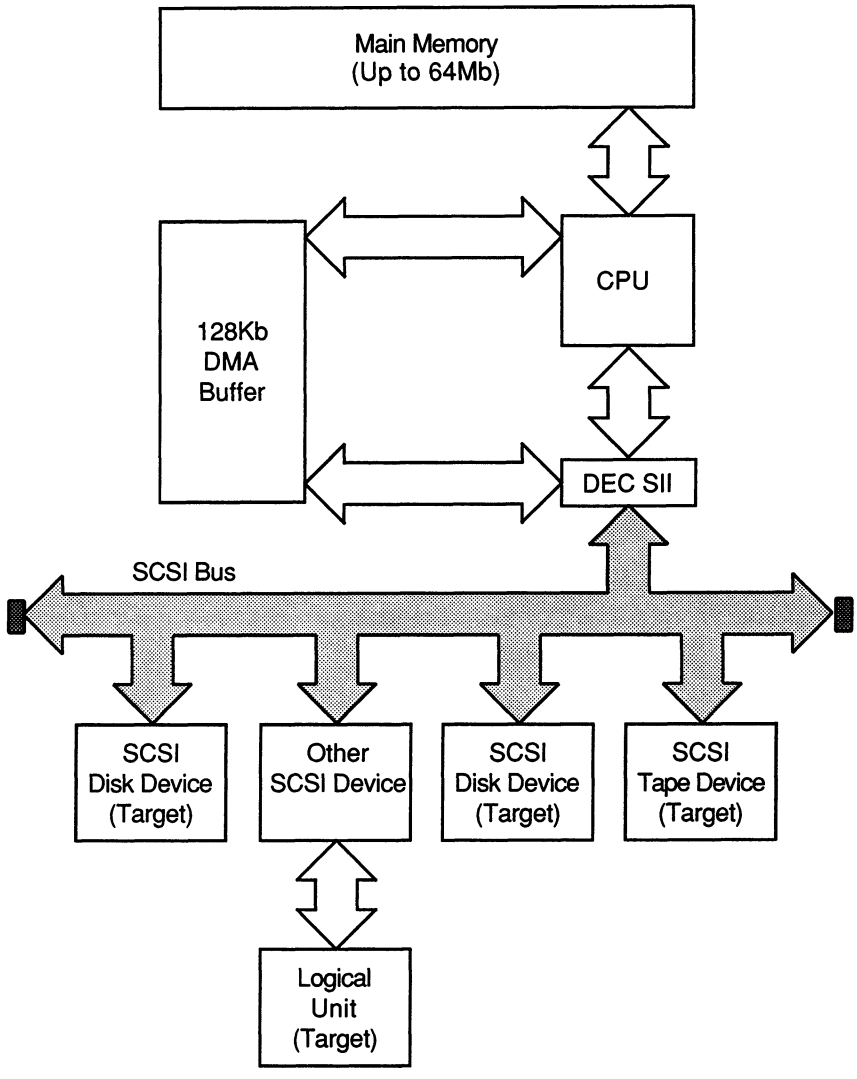
Figure 1–6 MicroVAX/VAXstation 3100 System Architecture



ZK–1367A–GE

The VAXstation 3520/3540 systems are multiprocessing systems, providing access to a single SCSI bus by means of Digital's SII SCSI controller chip. The SII chip supports both asynchronous and synchronous data transfers. VAXstation 3520/3540 systems do not support the SCSI asynchronous event notification feature. Figure 1–7 shows a representative configuration of the VAXstation 3520/3540 system.

Figure 1–7 VAXstation 3520/3540 System Architecture



Although VAX VMS SCSI implementation is currently based on the SCSI–1 standard, the SCSI–1 standard is upwardly compatible with SCSI–2. SCSI–2 clarifies many of the details specified in the SCSI–1 standard. Any non-Digital-supplied device to be attached to the SCSI bus of a MicroVAX/VAXstation system must implement all mandatory features of the SCSI–2 standard as described in the specification. The device is permitted to implement any optional features, as long as they are implemented according to the SCSI–2 standard. The device may implement vendor-unique features, as long as they are implemented in areas clearly designated as such by the standard.

The ANSI SCSI specification is, in places, very broad and flexible. In some cases, it is possible for a SCSI device to conform to the specification but be unsupported by the VMS operating system. For instance, it is possible that a SCSI device may implement a maximum timeout value that is incompatible with a value required by the VMS operating system. For additional information on VMS SCSI device support, see Chapter 17.

1.7 Programmed-I/O and Direct-Memory-Access Transfers

Devices are equipped with various registers that initiate, control, and monitor the progress of data transfer, seek operation, or other requests for device activity. When it completes a request, the device posts an interrupt to the processor. The size of the transfer concluded by a device interrupt depends upon the capabilities of the device.

1.7.1 Programmed I/O

Drivers for relatively slow devices, such as printers, card readers, terminals, and some disk and tape drives, must transfer data to a device register a byte or a word at a time. These drivers must themselves keep a record of the location of the data buffer in memory, as well as a running count of the amount of data that has been transferred to or from the device. Thus, these devices perform **programmed I/O** (PIO) in that the transfer is largely conducted by the driver program.

Examples of UNIBUS devices that do PIO transfers are the LP11 and the DZ11. Corresponding Q22 bus devices that perform PIO transfers are the LPV11 and the DZV11.

Chapter 2 outlines the action of the LP11 driver. The LP11 driver transfers data from a system buffer to the line printer data buffer register a byte at a time, while maintaining a count of the number of bytes left to transfer. When the line printer data buffer is full, the line printer sets a "not ready" bit in its status register. If the driver, while examining this register, sees this bit set, it enables interrupts from the printer and then suspends itself in the expectation that the printer will post an interrupt to the processor. While the driver remains suspended, the printer prints the data from its buffer and interrupts the processor when it is done. With the interrupt handled by the system interrupt dispatcher and the driver interrupt service routine, driver execution resumes. The driver repeats both its byte-by-byte transfer to the printer data buffer, as well as the entire routine described previously, until it determines that all the data has been transferred as requested.

Drivers performing PIO transfers are generally not concerned with the operation of I/O adapters. However, drivers that perform direct-memory-access (DMA) transfers must take into account I/O adapter functions, as discussed in Section 1.7.2.

1.7.2 Direct-Memory-Access I/O

Devices that perform **direct-memory-access** (DMA) transfers do not require the central processor so frequently. Once the driver activates the device, the device can transfer a large amount of data without requesting an interrupt after each of the smaller amounts. The responsibilities of a driver for a DMA device involve supplying a device register with the starting address of the buffer containing the data to be transferred, a byte offset into the buffer, and the size of the transfer. By setting the appropriate bit or bits in the device control and status register (CSR), the driver activates the device. The device then automatically transfers the specified amount of data to or from the specified address. The VMS drivers DLDRIVER and XADRIVER are examples of DMA drivers. They appear in full in the appendixes of the *VMS Device Support Reference Manual.*

For DMA transfers, UNIBUS and Q22 bus drivers must first map the transfer from main memory to I/O bus memory space. The result of this mapping is a set of contiguous addresses in UNIBUS or Q22 bus space that the DMA device can access to successfully perform a DMA transfer. To accomplish this, a driver must first obtain map registers, and, optionally for UNIBUS drivers, a buffered data path. The driver calls VMS routines that interface with the I/O adapter to allocate these resources on behalf of the driver. Chapter 14 discusses the operation of the UNIBUS adapter and the Q22 bus. Section 14.2 provides instructions on how to write a DMA driver for UNIBUS and Q22 bus devices.

Some controllers that can do DMA transfers on the Q22 bus have microcode that allows the controller itself to do physical-to-virtual address mapping. This allows such controllers to do scatter-gather mapping, eliminating the need for transfers to be made to or from physically contiguous main memory.

The method by which a generic VAXBI device capable of DMA transfers accomplishes such a transfer depends upon the characteristics of the device. Several methods are discussed in Section 16.6.

1.8 Buffered and Direct I/O

A separate issue, but one related to the data transfer capabilities of a device, results from the fact that the original buffer, as specified in the user $QIO request, is in process space and is mapped by process page-table entries. Because the driver cannot rely on process context existing at the time the device is ready to service the I/O request, it must have some means of guaranteeing that it can access both the data involved in the transfer and the page-table entries that map the buffer.

The VMS operating system provides the following two techniques that are employed by device drivers:

- **Direct I/O**, the technique used most commonly by drivers of DMA devices, locks the user buffer in memory as well as the page-table entries that map it. The function decision table (FDT) of such a driver calls a VMS-supplied FDT routine that prepares the user buffer for direct I/O.

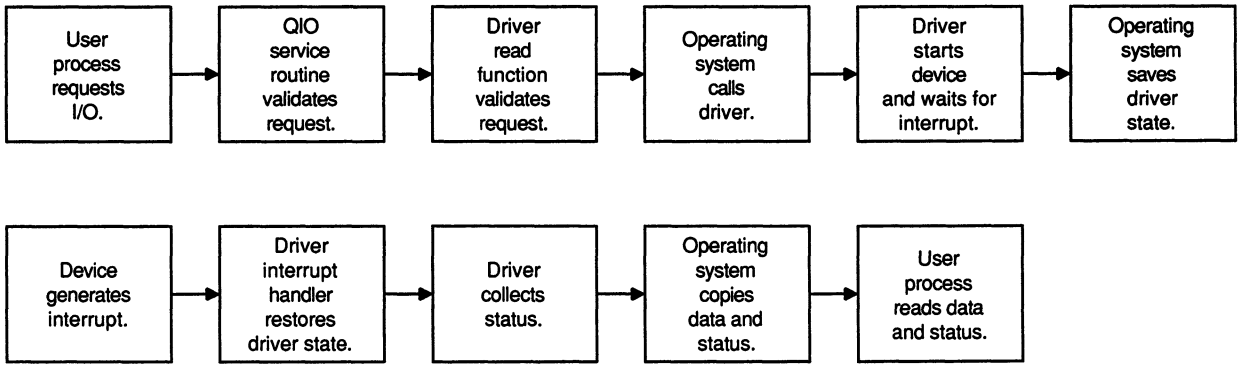
- **Buffered I/O** is the strategy whereby the driver FDT dispatches to an FDT routine in the driver that allocates a buffer from nonpaged pool. It is this intermediate buffer that is involved in the transfer. The driver later refers to the buffer using addresses in system space. Driver preprocessing routines copy the data from the user buffer to the system buffer for a write request; VMS I/O postprocessing (by means of a special kernel-mode AST) delivers data from the system buffer to the user buffer for a read request. Drivers most often use buffered I/O for PIO devices such as line printers and card readers.

The trade-off between buffered I/O and direct I/O is the time required to move the data into the user's buffer as against the time required to lock the buffer pages in memory. Sections 6.3.1 and 7.4 provide additional information.

1.9 Example of an I/O Request

Figure 1–8 illustrates how the VMS operating system and the device driver process a user request for a read I/O operation for a DMA device attached to a UNIBUS or Q22 bus.

Figure 1–8 Example of I/O Request Processing



ZK–0909–GE

The processing of the sample I/O request illustrated in Figure 1–8 occurs in the following steps:

1 **A process requests an I/O operation.**

A user process initiates an I/O request by issuing either a $QIO system service call or an RMS call resulting in a call to the $QIO system service.

The user process specifies the target device, a read function code, and the address of a buffer into which the data is to be read.

2 **The operating system performs I/O preprocessing.**

The $QIO system service validates the request and locates data structures in the I/O database that describe the device and its driver. The system service also allocates and initializes an I/O request packet to contain a description of the I/O request. The system service then calls a reading routine in the driver.

3 **The driver performs I/O preprocessing.**

The driver FDT routine verifies that the user buffer resides in virtual memory pages that can be modified by the requesting process, locks the buffer pages in memory, and adds details of the I/O operation to the I/O request packet. The read FDT routine then calls the operating system to send the I/O request packet to the driver.

4 **VMS creates a driver's fork process.**

A VMS routine creates a fork process in which the device driver can execute. The routine activates the driver's fork process by transferring control to the driver's start-I/O routine.

5 **The driver readies the I/O adapter.**

For DMA transfers, the driver's fork process calls VMS routines that enable the I/O adapter hardware to map I/O bus addresses into physical addresses for the transfer.

6 **The driver activates the device.**

The fork process activates the device by setting bits in device registers.

7 **The driver waits for an interrupt.**

A VMS routine saves the context of the driver's fork process and relinquishes the processor until an interrupt occurs.

8 **The device requests an interrupt.**

When the data transfer is complete, the device requests a hardware interrupt that causes the system to dispatch to the driver's interrupt service routine.

9 **The driver services the interrupt.**

The driver's interrupt service routine handles the interrupt and reactivates the driver, which reads device registers to obtain status information about the transfer.

10 **The operating system inserts the driver in a fork queue.**

The driver requests that it again be suspended, to be reactivated later at a lower software interrupt priority level (IPL).

11 **The fork dispatcher reactivates the driver's fork process.**

When processor priority permits, the VMS fork dispatcher reactivates the driver as a fork process.

12 **The driver completes the I/O operation.**

The driver's fork process completes device-dependent processing of the I/O request and returns the I/O status to VMS.

13 VMS completes the I/O operation.

The VMS I/O postprocessing routines copy the I/O status into process address space, general registers, or both, and return control to the user process.

Only four of these 13 steps describe the driver's I/O preprocessing and fork processing. The VMS I/O-support routines perform I/O processing common to many I/O requests. Driver writing is further simplified by the use of VMS routines that handle device-independent functions.

This example simplifies the processing of an I/O operation by ignoring such issues as

- The association of a device with a process, which is to say, device assignment
- Simultaneous I/O requests for one device
- System synchronization issues, such as IPLs and spin locks
- Driver competition for shared system and I/O adapter resources
- Driver competition for a multiunit controller
- Driver recovery from device errors or power failure

Subsequent chapters discuss each of these issues in relation to device drivers.

2 Discussion of a $QIO Request

This chapter outlines the series of activities performed by the VMS operating system and a simple device driver in order to process an I/O request. The LP11 line printer driver (LPDRIVER) was selected for this discussion because it is a simple driver but still illustrates many driver principles. The first-time reader of this document might not understand all of the points made in this chapter; however, the chapter should provide some insight into driver flow and I/O processing.

The LP11 printer is a programmed I/O (PIO) device (see Section 1.7.1). Although the LP11 is usually spooled, this discussion assumes it is not.

A user process can request the following functions on this printer:

- Write data to the printer
- Read the printer's device characteristics
- Alter the printer's device characteristics

This chapter describes two aspects of printer I/O processing:

- The portions of the line printer driver that are used in servicing a write request
- The VMS components with which the driver interacts to process the write request

Figure 2–1 illustrates the flow of execution through the VMS executive routines and printer driver code that satisfies an I/O request. Boxes above the solid line indicate processing in user-process context. Boxes below the line indicate processing in fork or interrupt context.

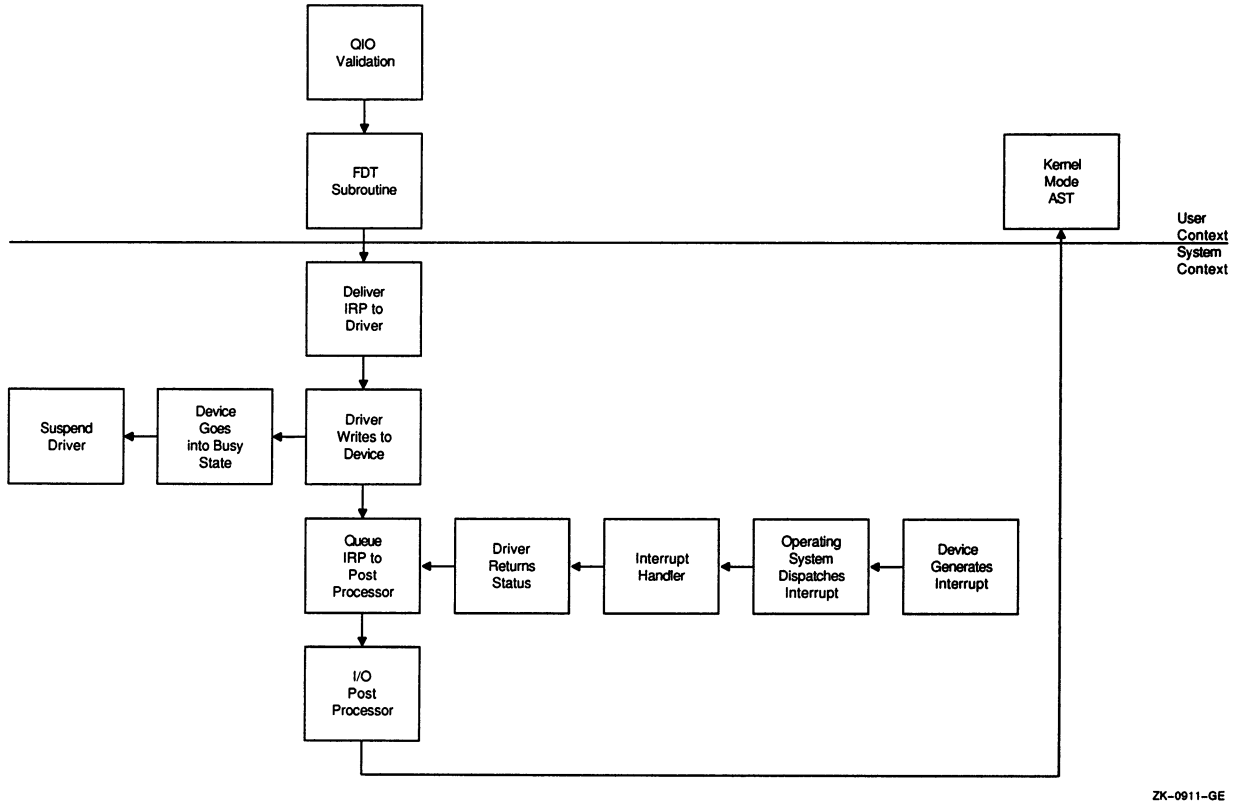
2.1 Driver Code for the LP11 Write Function

The VMS device driver for an LP11 printer implements a write function using the following parts of the driver:

- A function decision table (FDT) routine that reformats the user-supplied data
- A start-I/O routine that writes data to the printer data buffer register until the printer enters a busy state as it prints the contents of its internal print silo
- Code that modifies a device register to enable interrupts from the printer

- An interrupt service routine that returns control to the driver's fork process after a hardware interrupt from the printer
- Code that returns I/O status to a VMS I/O completion routine

Figure 2–1 A Printer Write Function



2.2 A User Process I/O Request

A user process writes a line to the printer by calling the Queue I/O Request ($QIO) system service, specifying the write-virtual-block function code as follows:

```
$QIO_S      chan = CHANNEL_NUMBER,-
            func = #IO$_WRITEVBLK,-
            efn = #6,-
            iosb = STATUS_BLOCK,-
            p1 = BUFFER_ADDRESS,-
            p2 = #BUFFER_SIZE,-
            p4 = #^X30      ;Carriage control character
```

Note that **p1**, **p2**, and **p4** are device-dependent arguments.

2.3 Device-Independent I/O Preprocessing by VMS

The $QIO system service first validates that the I/O request is correctly specified. The I/O request must meet the following criteria:

- The location CHANNEL_NUMBER must contain a number that serves as a valid index into the process's channel list. This means that the process must have previously assigned the printer to this process channel using the Assign I/O Channel ($ASSIGN) system service. Once $QIO locates the assigned channel control block, it can retrieve the address of the unit control block (UCB) of the target device of the request. Ultimately, it obtains the address of the driver's function decision table (FDT), by way of a chain of longword pointers within the I/O database:

  CCB → UCB → DDT → FDT address

- The driver FDT must list IO$_WRITEVBLK as a valid function for the device.
- The event flag number must be valid.
- The process's remaining buffered I/O count (BIOCNT) must permit the $QIO system service to perform a buffered-I/O request.
- The process must have write access to location STATUS_BLOCK, specified in the request for use as an I/O status block (IOSB).

If all of these checks succeed, the $QIO system service creates an I/O request packet (IRP) in nonpaged system address space. The service then writes all known details about the I/O request into the IRP.

If the target device for the I/O request is not file structured, the $QIO system service changes any virtual-function code to its equivalent logical-function code when it builds the IRP. Thus, for a printer device, IO$_WRITEVBLK is translated to IO$_WRITELBLK.

2.4 Device-Dependent I/O Preprocessing by the Driver

Once it has validated the I/O request, the $QIO system service scans the FDT for an entry that associates the IO$_WRITELBLK function code with an FDT routine. The system service calls the routine, which in the case of the printer driver is a device-specific routine located in the printer device driver.

The FDT routine confirms that the requesting process has read access to the buffer starting at BUFFER_ADDRESS. Then, the FDT routine buffers data from the process address space into system address space in the following steps:

1. It calculates the length of the required system space buffer.
2. If the job byte count quota for buffered I/O (JIB$L_BYTCNT) permits, the routine allocates a buffer from system address space, stores the address of the buffer in the IRP, and decreases the current job byte count quota.

3 It then synchronizes access to the printer's UCB by obtaining its mutex semaphore (UCB$L_LP_MUTEX) for write access. It can thus reliably preprocess the write request, depending upon information contained in the UCB.

   By obtaining the line printer mutex semaphore, the driver FDT routine effectively prevents processes active in a VMS multiprocessing system from initiating simultaneous functions on the printer. Also, in a VMS uniprocessing system, this action prevents contention between a process that has allocated the printer (and has been preempted in the midst of a write function) and any of its subprocesses that, when scheduled, may attempt to start a concurrent function that alters device characteristics.

4 It reads the description of the printer's current line and page position from the device's UCB.

5 It reformats the data from the process buffer into the system buffer, adding carriage-control characters, as specified in argument **p4** to the I/O request, before and after the data.

   Formatting includes such functions as the replacement of horizontal tabs with multiple spaces and the replacement of lowercase characters with uppercase characters, if necessary.

6 It rewrites updated line and page positions into the device's UCB. This information indicates what the current location on the page being printed will be when the request completes.

7 Finally, the routine transfers control to a VMS routine that queues the IRP to the device driver.

All of the I/O processing described to this point occurs in the context of the user's process. The user address space is mapped, and the processor's IPL is still low enough to permit process scheduling and paging. Subsequent queuing of the transfer request to the driver and all resulting driver processing occur at higher IPLs—and with ownership of the appropriate fork lock and device lock in a VMS multiprocessing environment—that synchronize the driver's handling of the device. (See Chapter 3 for a discussion of the concept of synchronization.)

2.5 Queuing the I/O Request Packet to the Driver

Before queuing the IRP to the printer driver, the VMS queuing routine raises the IPL to the driver's fork level and obtains the associated fork lock in a VMS multiprocessing environment. These actions synchronize access to those fields of the UCB referenced by driver routines at fork IPL.

If the device is idle, which is to say that if the busy bit in the UCB status longword (UCB$V_BSY in UCB$L_STS) of the UCB is clear, VMS can transfer control to the driver. The driver dispatch table (DDT) contains the entry point to the driver's start-I/O routine. To find the proper entry point, the queuing routine chains through the I/O database to the DDT, as follows:

UCB → DDT → start-I/O routine address

If the device unit is busy with another transfer, VMS inserts the IRP in a queue of packets waiting for the unit. The UCB contains the head of the queue. The packet's position in the queue depends on the scheduling priority of the process issuing the request.

2.6 Activating the Printer

The LP11 printer controller accepts data into an internal print silo until the silo is full or the driver writes a carriage-control character to the printer's data buffer register. When either event occurs, the printer sets a busy bit in the device's control and status register (CSR). Then the device driver sets the interrupt-enable bit in the device's CSR and waits for the printer to interrupt. When the printer requests a hardware interrupt, the driver can resume writing characters to the printer's data buffer register.

The driver routine delivers characters to the printer according to the following sequence:

1 The driver clears the device lock and locates the LP11 device registers using a chain of pointers starting at the device's UCB.

UCB → CRB → IDB → CSR address

The CSR address is always the address of the printer's CSR, and all other device registers are at fixed offsets from this address. In contrast to many other devices, such as disks, the LP11 printer does not share a controller with other devices; therefore, no arbitration for ownership of the controller is required.

2 The driver examines the device's CSR to see if the device is ready to accept characters.

3 If the device is ready, the driver writes a byte of data to the printer's data buffer register. The printer controller moves the byte from the register to the controller's internal print silo.

4 The driver decreases the count of bytes to transfer and repeats step 2.

5 If the device is not ready (that is, its print silo is full), the driver raises IPL to device IPL and obtains the corresponding device lock in a VMS multiprocessing system. These actions allow it to set the interrupt-enable bit in the device's CSR in synchronization with other routines in the driver that may access the CSR.

After setting the interrupt-enable bit, the driver invokes a VMS wait-for-interrupt macro to release the device lock and suspend driver processing until the printer requests an interrupt or the device times out.

2.7 Waiting for a Device Interrupt

The VMS wait-for-interrupt routine suspends the driver by performing the following functions:

- Saving driver context (R3, R4, and the address of the next instruction in the driver) in the device's UCB

- Calculating the time at which the device will time out
- Setting bits in the device's UCB to indicate that the driver expects a device interrupt within a specified time period
- Releasing the device lock in a VMS multiprocessing system, restoring IPL to fork level, and returning control to the caller of the driver's start-I/O routine

The driver remains in a suspended state until one of the following two events occurs:

- The printer requests a hardware interrupt.
- VMS reports a device timeout because the printer did not request a hardware interrupt within a specified period of time.

Normally, the LP11 prints the contents of its data buffer and requests the interrupt.

2.8 Handling Interrupts

When the LP11 printer requests a hardware interrupt, the interrupt dispatcher passes the interrupt to the LP11 driver's interrupt service routine.

The driver's interrupt service routine restores control to the driver, as follows:

1. Restores the address of the UCB in R5
2. Obtains the appropriate device lock to ensure synchronization in a VMS multiprocessing environment
3. Confirms that the interrupt was expected by examining bits in the device's UCB
4. Restores the saved registers (R3 and R4) from the device's UCB
5. Transfers control to the driver program counter (PC) address stored in the device's UCB

Rather than execute in interrupt context, the reactivated driver routine calls a VMS routine to create a fork process. As a result of this action, VMS again suspends driver processing by performing the following steps:

1. Saving driver context (R3, R4, and the driver PC address) in the device's UCB
2. Inserting the UCB address in the appropriate fork queue in the local processor's CPU database

The driver suspension allows the operating system to reschedule driver processing at its fork IPL and permits higher priority code to execute and device interrupts to be serviced while driver processing of the I/O request concludes. The VMS fork dispatcher reactivates the driver when the IPL of the local processor drops to fork level.

After creating the fork process, the system returns control to the driver's interrupt service routine, which restores the registers saved at the time of the device interrupt, releases the device lock, and dismisses the interrupt.

2.9 I/O Postprocessing by the Driver

When the VMS fork dispatcher reactivates the driver's fork process, the driver obtains the number of characters left to transfer from the UCB. If there are still characters to transfer, the driver and printer repeat the procedures outlined in Sections 2.6 to 2.8, until the transfer is complete. When all characters have been transferred, the driver code branches to the driver's I/O-completion code.

The driver's I/O-completion code stores a success status code and the number of bytes transferred in R0, then transfers control to VMS to complete the I/O request.

2.10 I/O Postprocessing by VMS

The operating system inserts the IRP into the systemwide I/O postprocessing queue and requests an interrupt from the processor at IPL$_IOPOST. If another IRP is queued to the UCB for the device unit, VMS dequeues that packet and calls the driver start-I/O routine to process it. When IPL drops to IPL$_IOPOST, the processor grants the I/O postprocessing interrupt request. The I/O postprocessing dispatcher dequeues the packet for the printer I/O request and performs the following steps:

1. Increases the use count (PHD$L_BIOCNT) of the process's buffered I/O requests because the current operation is complete. The use count is maintained for accounting purposes.
2. Decreases the process's buffered I/O count (PCB$W_BIOCNT) to reflect a completed buffered I/O operation. This operation restores buffered-I/O quota to the process.
3. Deallocates the system buffer used for the reformatted user data.
4. Increases the job's byte count quota.
5. Sets an event flag to indicate that the I/O operation is complete.
6. Queues a special kernel-mode asynchronous system trap (AST) routine that will deallocate the IRP and stores I/O status in the user's I/O status block (IOSB).

The user process determines when the I/O operation is complete by the setting of the event flag or the filling of the (IOSB), or both, according to the method defined in the I/O request. The Queue I/O Request and Wait ($QIOW) system service completes synchronously and returns control and status to the user process only after the I/O operation has been completed. The Synchronize ($SYNCH) system service waits for the completion of an I/O request, initiated by the $QIO system service, that completes asynchronously to user process activity.

3 Synchronization of I/O Request Processing

Because a device driver executes as kernel-mode code, it can preempt core system tasks and access critical system data. As a result, it must adhere to a set of rules that governs the priority of system activities and controls the flow of system events. These synchronization rules ensure that both the operating system and the device driver access memory in an orderly and consistent fashion.

This chapter contains the following discussions:

- Section 3.1 discusses the interrupt priority levels (IPLs), focusing on those IPLs and interrupt service routines that participate in the processing of an I/O request. It briefly examines the roles of the other IPLs in the operating system. Whether you are writing a driver for a VMS uniprocessor or multiprocessor environment, you must adhere to the synchronization rules discussed in this section.
- Section 3.2 discusses the various VMS spin lock semaphores, including when and how they ensure proper multiprocessing operation.
- Section 3.3 illustrates how system synchronization is maintained during the processing of an I/O request on any VAX system. As part of this discussion, this section describes the driver fork process and the activity of forking. Finally, it examines the methods by which a driver synchronizes at fork level and device interrupt level.
- Section 3.4 discusses the mechanism by which driver code stalls to wait for an available adapter or controller resource on any VAX system.

3.1 Interrupt Priority Levels

The VAX architecture defines 32 levels of hardware priority, called interrupt priority levels (IPLs). These IPLs govern the sequence of system events that occur on each processor in a VAX system. The higher-numbered IPLs (16 to 31) are reserved for hardware interrupts, and the lower-numbered IPLs (1 to 15) are reserved for software interrupts. Most process-based software runs at IPL 0.

The hardware IPLs (16 to 31) are used for device interrupts (IPLs 20 to 23), interprocessor interrupts in a multiprocessing system, interval timer interrupts, urgent conditions like power failure, and such serious errors as a machine check. Those IPLs that have a bearing on driver execution are discussed in Sections 3.1.2 and 3.1.3. For specific hardware IPL information, see your VAX system's hardware documentation or the *VAX Hardware Handbook*.

Synchronization of I/O Request Processing

3.1 Interrupt Priority Levels

The software IPLs (1 to 15) are defined by VMS as illustrated in Table 3–1.

Table 3–1 IPLs Defined by VMS

IPL	Symbolic Name	Use
0	—	Execution of most process-based software
1	—	Reserved
2	IPL$_ASTDEL	Servicing of AST-delivery interrupts
3	IPL$_RESCHED	Servicing of scheduler interrupts
4	IPL$_IOPOST	Servicing of I/O-postprocessing interrupts
5	—	Reserved
6	IPL$_QUEUEAST	Fork level processing for queuing ASTs
7	IPL$_TIMERFORK	Entry level for software timer interrupt servicing
8	IPL$_SYNCH	Synchronization of access-to-system databases in a uniprocessor system[1]
11	IPL$_MAILBOX IPL$_POOL	Fork level processing for access to mailboxes Allocation of nonpaged pool
8–11	—	Fork level processing for executing driver code
12	—	Recalculation of quorum; cancellation of mount verification (IPC)
13	—	Reserved
14	—	Entry level for XDelta debugger
15	—	Reserved

[1] IPL$_TIMER, IPL$_SCHED, IPL$_JIB, IPL$_MMG, IPL$_FILSYS, and IPL$_IOLOCK8 are all synonyms for IPL$_SYNCH (see Table 3–3).

Because a higher IPL takes precedence over a lower IPL, a routine executing at one IPL can block interrupts on a processor at that IPL and all lower IPLs. This scheme allows VMS to assign the higher IPLs to system activities that must be dispatched quickly and with little chance of interruption. In a general sense, each processor services interrupts according to the following priorities:

- Power failure
- Processor errors
- Device interrupts
- Device driver fork processing
- I/O postprocessing
- Process rescheduling
- AST delivery

As a result of blocking events on and ordering the activities of a single VAX processor, VMS use of IPLs ensures that kernel-mode code accesses data in memory in a cooperative and predictable manner. The mechanism by which synchronized access to data is ensured is twofold. First, VMS associates a given IPL with the access of one or more data structures or databases. Secondly, VMS defines an ordered set of semaphores, called **spin locks**, that extend IPL synchronization throughout a VMS multiprocessing system. A processor must obtain one or more of these spin locks before executing any code thread that must make use of the resources the spin lock protects. Spin locks thus allow each processor in a VMS multiprocessing system to share common system data and block events systemwide.

For example, consider a code thread running at IPL 4 that intends to access the memory management database. To do so, it raises IPL to IPL$_MMG. This action gives it the exclusive right to access the database from the local processor, effectively preventing access by other code threads on the same processor. After raising IPL, this code thread requests the memory management (MMG) spin lock. Ownership of the MMG spin lock gives the processor executing this thread the exclusive right to access the database systemwide, and bars access from any other code thread running on any other processor in the VAX system.

Although discussions in this book treat IPL and spin lock synchronization as conceptually separate tasks for a device driver, VMS synchronization macros make adjustment of IPL and disposition of spin locks appear as a single operation.

A full description of spin locks appears in Section 3.2.

3.1.1 Interrupt Service Routines

VMS associates certain IPLs with the execution of certain tasks. Moreover, when a processor in a VAX system grants an interrupt at a given IPL, the grant actually triggers the execution of a specific piece of code, called an interrupt service routine, that performs the task.

Device drivers themselves contain an interrupt service routine that handles device interrupts at an appropriate device IPL (IPLs 20 to 23). In addition, drivers rely heavily upon the VMS interrupt service routine known as the **fork dispatcher** that runs at several IPLs, including driver fork IPLs 8 to 11. When the local processor's IPL drops to fork IPL, it is the fork dispatcher that restores the context of the driver fork process and places it into execution. (See Sections 3.1.2.4 and 3.3.2 for discussions of the device IPLs and interrupt dispatching, respectively. Sections 3.1.2.3 and 3.3.3 discuss the fork IPLs and driver fork processes.)

3.1.2 IPL Use During I/O Processing

The activities essential to the processing of an I/O request occur only at certain IPLs. VMS performs some of these tasks in system routines and interrupt service routines; drivers perform others. This section describes those IPLs and interrupt service routines that are most involved in I/O processing. Section 3.1.3 discusses the IPLs at which other system activities transpire that may influence the coding of a driver. For additional information on the pattern of synchronization throughout the servicing of an I/O request, see Section 3.3.

3.1.2.1 IPL 2 (IPL$_ASTDEL)

The asynchronous system trap (AST) delivery interrupt service routine (SCH$ASTDEL) is associated with IPL$_ASTDEL.

When an AST is specified for delivery to a process, the AST queuing routine (SCH$QAST) queues the AST to the specified process's process control block (PCB).[1] When an AST is delivered is determined by the mode of the AST, the current mode of the processor, and the mode contained in the processor's ASTLVL register. The VAX hardware, by means of the REI instruction, requests a software interrupt on the local processor at IPL$_ASTDEL whenever the processor's mode becomes less privileged than that specified as its ASTLVL.[2]

The AST delivery interrupt service routine gains control when the processor's IPL drops below IPL$_ASTDEL, and delivers all deliverable ASTs to the currently scheduled process. Any code executing at IPL$_ASTDEL or higher blocks the execution of this interrupt service routine.

To block the delivery of ASTs—specifically the kernel-mode AST that causes process deletion—I/O preprocessing, from the time that the $QIO system service allocates an IRP through the execution of the last FDT routine, occurs at IPLs no lower than IPL$_ASTDEL. The VMS allocation routine records the address of the system memory allocated for the IRP in a process register; if an AST that deletes the process were to occur, the allocated memory would be lost from the pool.

In addition, some I/O postprocessing occurs in a special kernel-mode AST servicing routine that also executes at IPL$_ASTDEL. The special kernel-mode AST, running in the context of a process whose I/O has been completed, writes status information into an I/O status block, copies buffered input into process space, and deallocates system buffers. The completion of these tasks depends on the availability of process context.

[1] Because the VMS AST queuing and delivery routines access the scheduler database, they synchronize within a VMS multiprocessing environment by obtaining the SCHED spin lock before modifying system data.

[2] In the event that a processor queues an AST to a process currently executing on another processor in a multiprocessing system, the local processor generates an interprocessor interrupt to the other processor to change its ASTLVL.

Page faults may be taken by code that executes at IPL$_ASTDEL. However, this is not the case with code executing at higher IPLs. Thus, programs that are sensitive to the contents of pageable data structures run at IPL$_ASTDEL to take page faults. For example, the allocation of paged pool is one such program code thread; paged pool, as a result, is protected by a mutex.

3.1.2.2 IPL 4 (IPL$_IOPOST)

The IPL$_IOPOST interrupt service routine (IOC$IOPOST) performs device-independent postprocessing of an I/O request. As appropriate to the I/O request, it adjusts process quota use and deallocates system memory. IOC$IOPOST also queues a special kernel-mode AST to the process's PCB that, once process context is restored, writes status and data into the process's address space.

After it has completed whatever device-dependent postprocessing is required, a driver fork process requests I/O postprocessing by calling a VMS routine (IOC$REQCOM) that inserts an IRP in the systemwide I/O postprocessing queue and requests a software interrupt at IPL$_IOPOST. When IPL drops below IPL 4, the IPL$_IOPOST interrupt service routine dequeues an IRP from the I/O postprocessing queue, performs all I/O-completion tasks that can occur without reference to the device's unit control block (UCB) and, thus, at an IPL lower than fork IPL.

I/O postprocessing runs at an IPL higher than IPL$_RESCHED so that all pending I/O-completion processing is finished before the scheduler looks for a new process to schedule. The ability of a process to execute can depend on the completion of the postprocessing of an I/O request. Additionally, I/O postprocessing can queue ASTs to certain processes, thus changing their state to computable and resulting in a priority boost. Because all I/O completions are accomplished before rescheduling activities, the scheduler can select from a potentially larger set of computable processes, using more up-to-date information about these processes.

3.1.2.3 IPL 8 to IPL 11 (Fork IPLs)

On each processor in a VAX system—for each of the IPLs from 8 to 11—there exists a queue for fork blocks waiting to be processed. Each fork block contains the context of a suspended fork process. The interrupt service routine that executes at each of these IPLs (EXE$FORKDSPTH) is known as the **fork dispatcher**. The fork dispatcher dequeues a fork block, obtains the appropriate fork lock, restores the context of the fork process, and resumes its execution at the PC location saved in the fork block (at FKB$L_FPC). (Refer to Section 3.3.3 for a discussion of fork blocks and fork processes.)

All driver routines, except most FDT routines, execute at fork IPL or higher. Usually driver routines should not read or alter UCB fields without taking steps to ensure synchronization. Because such UCB fields can be shared among driver fork processes and VMS system tasks executing on other processors in a VMS multiprocessing system, a processor must first secure the corresponding fork lock to execute at that fork IPL. Furthermore, the drivers for all devices on a single I/O adapter must use the same fork lock if they actively compete for shared I/O adapter resources such as map registers and data paths. The VMS

routine that initiates an I/O request on an idle device unit, as well as the fork dispatcher, transfers control to the driver with the appropriate synchronization.

A driver places a fork lock index in UCB$B_FLCK using the DPT_STORE macro. (See Section 6.1.) VMS determines the appropriate fork IPL from the contents of the SPL$B_IPL field in the fork lock's structure. (See Section 3.2 for a discussion of spin locks.)

3.1.2.4 IPL 20 to IPL 23 (Device IPLs)

VAX peripheral devices request interrupts at IPLs 20 to 23 because device interrupts usually need to preempt most user and VMS software functions. When a device requests an interrupt at one of these IPLs and the processor is executing at a lower IPL, the processor grants the interrupt, and then transfers control to an interrupt service routine for the device located in its driver. If the processor is executing at a higher or equal IPL, the interrupt remains pending.

The **interrupt dispatcher** routes interrupts from devices to the appropriate device driver's interrupt service routine. A driver specifies the address of its interrupt service routine in the driver prologue table (DPT). The interrupt dispatcher's routing mechanism works differently depending upon the VAX processor and I/O subsystem in use. (For additional information on device interrupt dispatching, see the general discussion of interrupt dispatching in Chapter 9. Information specific to a given I/O subsystem configuration appears in Sections 14.3 (UNIBUS/Q22 bus), 15.4 (MASSBUS), and 16.4.1 (VAXBI bus).)

Data in a device's registers and in various fields of the UCB that record device status is synchronized on the local processor at device IPL, at which its driver's interrupt service routine executes. This value is stored by the driver in the UCB$B_DIPL field of the UCB. It is the responsibility of the interrupt service routine to secure the corresponding device lock. This action allows it to synchronize with other code threads that access the same resources in a VMS multiprocessing system.

The driver's start-I/O routine is one such code thread and must similarly synchronize. In a VMS uniprocessing environment, the routine raises IPL to device IPL before writing data in device registers and database fields. In a VMS multiprocessor environment, the start-I/O routine must secure the appropriate device lock to achieve systemwide synchronization of the device database. The act of acquiring the device lock automatically sets IPL to device IPL.

Because code executing at IPLs 20 to 23 blocks most other hardware interrupts and all software interrupts on the local processor, driver code lowers its IPL as soon as possible. Interrupts from devices on a MicroVAX system, VAX 82*x*0/83*x*0, VAX 85*x*0/8700/88*x*0, VAX 6000-series, or a VAX 9000-series system, in fact, can block hardware interrupts from the processor's interval clock if these device interrupts occur at or above IPL 22. To prevent the loss of an interval clock interrupt, these drivers, when executing at IPL 22 or above, should lower IPL below 22 as soon as possible (within 9 milliseconds).

3.1.2.5 IPL 31 (IPL$_POWER)

The highest IPL, IPL$_POWER (IPL 31), locks out all other interrupts on the local processor. Many VMS routines and drivers raise IPL to IPL$_POWER to execute code sequences that cannot tolerate interruption. For example, much of system initialization occurs at IPL$_POWER. In a VMS multiprocessing system, these routines often need to acquire additional synchronization, as described in Section 3.2.

When a device driver needs to execute a series of instructions without interruption, the driver raises IPL to IPL$_POWER. The driver should never remain at IPL$_POWER for more than a few instructions. The most common instance of a driver's raising IPL to IPL$_POWER is to determine whether a power failure has occurred on the local processor between the time that the driver writes setup data into device registers and the time that the driver starts the device by writing into the device's control register.

3.1.3 Additional IPLs

In addition to the IPLs that are directly involved in the processing of an I/O request, VMS defines the IPLs described in this section.

3.1.3.1 IPL 3 (IPL$_RESCHED)

When an event occurs that requires that a process be rescheduled, a VMS routine requests a software interrupt on the local processor at IPL$_RESCHED. The scheduler interrupt service routine (SCH$RESCHED) gains control at this IPL, but immediately obtains the SCHED spin lock (as a result, raising IPL to IPL$_SYNCH). This action synchronizes the processor's access to the scheduler's database with other system activities.

Drivers never explicitly reference IPL$_RESCHED. Most driver processing occurs at higher IPLs. When a process raises IPL to or above IPL$_RESCHED, the scheduler cannot reschedule the process. The process runs until an interrupt occurs at a higher IPL or the process lowers IPL below IPL$_RESCHED.

3.1.3.2 IPL 6 (IPL$_QUEUEAST)

IPL$_QUEUEAST is a fork-level IPL used predominantly by drivers written before Version 4.0 of the VMS operating system. A driver fork process originating at an IPL between 8 and 11 would use IPL$_QUEUEAST when it needed to synchronize access to the scheduler's database at IPL$_SYNCH—for instance, to queue an AST. Prior to VMS Version 4.0, the only way that such a fork process could maintain proper synchronization was to first call a system routine that creates yet another fork process to be dispatched at IPL$_QUEUEAST. Once the fork dispatcher dequeued the fork block and resumed execution of the driver, the driver fork process could then raise IPL to IPL$_SYNCH and access the system database.

Because versions of the VMS operating system after Version 4.0 implement IPL$_SYNCH as a fork IPL, a driver fork process can fork directly to IPL$_SYNCH. Since VMS Version 5.0, the fork dispatcher obtains the IPL 8 fork lock (IOLOCK8), dequeues the driver fork block, restores driver

context, and resumes execution of the driver. To maintain synchronization in a VMS multiprocessing environment, the driver then must obtain the spin lock that corresponds to the data structure it is accessing.

3.1.3.3 IPL 7 (IPL$_TIMERFORK)

The interval clock's interrupt service routine (EXE$HWCLKINT), executing at IPL 22 or IPL 24 depending upon the VAX system, posts interrupts at IPL$_TIMERFORK. A processor requests such an interrupt when the current process has exceeded its processor time quantum. The software timer interrupt service routine (EXE$SWTIMINT) gets control when the IPL drops below IPL$_TIMERFORK, services quantum end events by immediately raising IPL to IPL$_SYNCH (obtaining the SCHED spin lock, if needed), and calls the appropriate scheduler routine.

The primary processor in a VMS multiprocessor system, when executing the interval clock's interrupt service routine, requests an IPL$_TIMERFORK interrupt when the first entry in the timer queue (EXE$GQ_1ST_TIME) is due. The software timer interrupt service routine contains special code that allows the primary processor to service the expiration of a timer queue element (TQE). The routine raises IPL to IPL$_SYNCH, synchronizes access to the timer queue (except for the first TQE) by obtaining the TIMER spin lock, and secures the interval clock database (the system time at EXE$GQ_SYSTIME and the expiration time of the first TQE at EXE$GQ_1ST_TIME) by obtaining the HWCLK spin lock. Thus synchronized, it determines which TQEs have expired, dequeues them, and transfers control to the appropriate timeout handlers. Device timeouts are dispatched in this manner.

3.1.3.4 IPL 8 (IPL$_SYNCH)

IPL$_SYNCH is the level at which the databases that record and control system functions are synchronized. Individual spin locks, such as the JIB, SCHED, MMG, and TIMER spin locks, provide synchronized access to individual databases in a VMS multiprocessing environment.[3] When a VMS subroutine or a driver needs to modify or read a dynamic portion of a system database, the routine always executes at IPL$_SYNCH, holding an appropriate system spin lock, to ensure that the database does not change because of some interrupt service routine or process action.

3.1.3.5 IPL 11 (IPL$_MAILBOX)

IPL$_MAILBOX is the highest fork IPL. When a VMS or driver routine writes into a mailbox, the executing processor must be at IPL$_MAILBOX holding the MAILBOX spin lock. Because other readers or writers to the mailbox must similarly pursue synchronization, these actions prevent other writers from modifying incomplete data in the mailbox and readers from reading invalid data.

[3] IPL$_TIMER, IPL$_SCHED, IPL$_SCS, IPL$_JIB, IPL$_MMG, IPL$_FILSYS, and IPL$_IOLOCK8 are all synonyms for IPL$_SYNCH (see Table 3–3).

3.1.3.6 IPL 14 (XDELTA Entry IPL)

For debugging purposes, you can halt a processor from the console terminal and request a software interrupt to invoke the XDelta debugger. You accomplish this by depositing $0E_{16}$ in the processor's Software Interrupt Request Register (PR$_SIRR). (The procedure for requesting a software interrupt to invoke XDELTA is described in Table 13–3.)

After you issue the console's CONTINUE command and return to program mode, the processor grants an interrupt at IPL 14. The processor must be executing below the requested IPL for the interrupt to take effect.

3.1.3.7 IPL 22 or IPL 24 (Interval Clock IPLs)

Every 10 milliseconds, the interval clock interrupts at IPL 22 or 24, depending upon the VAX system. A system cell points to the IPL field (SPL$B_IPL) in the HWCLK spin lock, identifying the IPL at which the processor's interval clock interrupts.

The interval clock's interrupt service routine performs the functions described in Section 3.1.3.3. Note that the interval clock's interrupt service routine obtains the HWCLK spin lock to synchronize its operations on the system time quadword (EXE$GQ_SYSTIME) and the quadword containing the due time of the first timer queue element (EXE$GQ_1ST_TIME).

3.1.4 Modifying IPL in Driver Code

Kernel-mode code can modify the IPL of the local processor by either explicitly setting the processor's IPL to a specific value or by requesting a software interrupt at a specific level. Driver code can change the IPL at which it executes by invoking a VMS-supplied macro to request a change in IPL. Because the DEVICELOCK, FORKLOCK, and LOCK macros (and their counterparts) raise (or lower) IPL in a VMS uniprocessing environment, and achieve full synchronization in a VMS multiprocessing system, Digital recommends their use instead of the SETIPL, DSBINT, and ENBINT macros.

Table 3–2 lists the macros that set, store, or restore a processor's IPL. See the macro chapter of the *VMS Device Support Reference Manual* for a further explanation of the functions of these macros and a full description of their arguments.

Table 3–2 VMS Macros That Change a Processor IPL

Macro	Function
Raising IPL	
DEVICELOCK *[lockaddr] [,lockipl] [,savipl] [,condition] [,preserve=YES]*	Raises IPL on the local processor to the device IPL associated with the device lock's *lockaddr*, obtains the device lock, and saves the current IPL at *savipl* [1]
DSBINT *[ipl=31] [,dst=–(SP)] [,environ=MULTIPROCESSOR]*	Raises IPL on the local processor to the specified *ipl*, saving the current IPL at *dst* [2]
FORKLOCK *[lock] [,lockipl] [,savipl] [,preserve=YES] [,fipl=NO]*	Raises IPL on the local processor to *lockipl*, obtains the fork lock, and saves the current IPL at *savipl* [1]
LOCK *lockname [,lockipl] [,savipl] [,condition] [,preserve=YES]*	Raises IPL on the local processor to the *lockipl*, obtains the lock indicated by *lockname*, and saves the current IPL at *savipl* [1]
SETIPL *[ipl=31] [,environ=MULTIPROCESSOR]*	Raises IPL on the local processor to the specified *ipl* [2]
Lowering IPL	
DEVICEUNLOCK *[lockaddr] [,newipl] [,condition] [,preserve=YES]*	Releases or restores the device lock indicated by *lockaddr*, lowering the local processor's IPL to *newipl*, thus permitting interrupts to occur at or beneath the current IPL[1]
ENBINT *[src=(SP+)]*	Lowers the local processor's IPL to *src*, thus permitting interrupts to occur at or beneath the current IPL[2]
FORKUNLOCK *[lock] [,newipl] [,condition] [,preserve=YES]*	Releases or restores the fork lock indicated by *lock*, lowering the local processor's IPL to *newipl*, thus permitting interrupts to occur at or beneath the current IPL[1]
UNLOCK *lockname [,newipl] [,condition] [,preserve=YES]*	Releases or restores the spin lock indicated by *lockname* and lowers IPL to *newipl*, thus permitting interrupts to occur at or beneath the current IPL[1]
Miscellaneous Functions	
SAVIPL *[dst=–(SP)]*	Saves the local processor's IPL at the specified location
SOFTINT *ipl*	Requests a software interrupt on the local processor at the specified *ipl*

[1]When used in a uniprocessing environment, the DEVICELOCK, DEVICEUNLOCK, FORKLOCK, FORKUNLOCK, LOCK, and UNLOCK macros generate only the code that manipulates IPL.

[2]Use of the SETIPL, ENBINT, and DSBINT macros is not sufficient to guarantee systemwide synchronization of events and data in a VMS multiprocessing system. The DEVICELOCK, FORKLOCK, and LOCK macros have been designed to achieve appropriate synchronization in either a uniprocessing or multiprocessing environment.

3.1.4.1 Raising IPL

To block certain activities on a local processor in a VAX system, it is sometimes useful to raise IPL explicitly. Driver code should not raise IPL for more than a few instructions, for doing so prevents the local processor from servicing interrupts at the current IPL and all lower IPLs.

In a uniprocessor environment, raising IPL provides sufficient systemwide synchronization to both block events and also protects data customarily accessed at a given IPL. Drivers typically raise a processor's IPL to check for a local processor power failure, to send a message to a mailbox, or to access device registers. For instance, a driver running exclusively in a VMS uniprocessor environment can set IPL to its device IPL (UCB$B_DIPL) to access device registers. While the driver executes at device IPL, no other code thread can execute at the same device IPL and thereby read or write the same device registers. (See Section 3.1.4.2 for a description of the rules for lowering IPL that enforce the synchronization.) VMS supplies the SETIPL and DSBINT macros to effect the change in IPL.

In a multiprocessing environment, as in a uniprocessing environment, a driver can block activities on the local processor by raising IPL. However, in a multiprocessing environment, simply raising IPL is not sufficient to protect shared data structures from other processors that may attempt to access them concurrently. To achieve synchronization in a VMS multiprocessing system, VMS associates a series of semaphores, called **spin locks**, with such shared databases. (See Section 3.2 for a further discussion of spin locks.)

A processor that must access a shared structure must first secure a corresponding spin lock. The acquisition of a spin lock often involves the raising of IPL to the IPL associated with the spin lock and the database it protects. Spin lock acquisition code can elevate IPL automatically if called from a code thread executing at an IPL lower than the synchronization IPL of the lock. A processor that has properly obtained a spin lock can thus proceed to access the associated database at the appropriate IPL. If necessary, it is free to further raise IPL, but should not lower IPL below the spin lock's allocation IPL without first releasing the spin lock.

For example, a driver running in a VMS multiprocessing system can set IPL to IPL$_POWER to block the servicing of a power failure on the local processor. However, while executing at IPL$_POWER (or at device IPL), the driver cannot safely access device registers unless it has first secured the spin lock associated with the device; that is, its device lock. Similarly, a driver's fork process, although it executes at a fork IPL with a corresponding fork lock held, may raise a processor's IPL by obtaining an additional spin lock. Sending a message to the OPCOM (operators communication process) mailbox (obtaining the MAILBOX spin lock at IPL 11) and accessing device registers (obtaining the appropriate device lock at device IPL) are two such activities.

The LOCK, FORKLOCK, and DEVICELOCK macros ensure that the synchronization needed for either the uniprocessor or multiprocessor environment is obtained before the requested resource is accessed. When executed in a uniprocessor environment, these macros only obtain the proper IPL synchronization. When invoked in a multiprocessing environment, these macros both raise IPL and obtain an appropriate spin lock, thus extending IPL synchronization systemwide.

3.1.4.2 Lowering IPL

Driver code lowers its IPL to synchronize with code threads that access common data or perform common activities at the lower IPL. In a multiprocessing environment, lowering IPL is often associated with the release of a spin lock. In addition, lowering IPL may be necessary in order to obtain a spin lock synchronized at the lower IPL.

One of the most fundamental coding rules in VMS is that *a code thread cannot explicitly lower IPL below the level at which its execution has been initiated.* In relation to driver processing, this means that a driver fork process cannot explicitly set IPL to be less than its fork IPL, nor can a driver's interrupt service routine explicitly set IPL to be less than device IPL. This is because a processor interrupted a lower IPL code thread in mid-execution to place the current code thread into execution. It is important to the integrity of the data structures protected at this lower IPL that the previous code thread be resumed before other code accesses the same structures. A violation of the IPL rule would undermine the VMS interrupt dispatching mechanism by not first returning control to the interrupted code thread.

Driver code uses the following methods to lower IPL:

- Issuing a DEVICEUNLOCK, FORKUNLOCK, or UNLOCK macro (paired with an earlier invocation of a DEVICELOCK, FORKLOCK, or LOCK macro) or a ENBINT macro (paired with an earlier invocation of an DSBINT macro) to restore IPL to a previously saved value.
- Invoking the IOFORK (or FORK) macro to preserve its context in a fork block, to insert the block in a fork queue, and to request a software interrupt at the driver's fork IPL. See Section 3.3.3.1 for a complete discussion of forking.
- Issuing an REI instruction at the end of its interrupt service routine that dismisses the interrupt.

Lowering IPL can cause many pending interrupts on the local processor between the old and new IPLs to become deliverable.

3.2 Spin Locks

In a multiprocessing environment, as in a uniprocessing environment, you can block activities on the local processor by raising IPL. Similarly, certain shared databases must be accessed only at a given IPL. However, in a multiprocessing environment, simply raising IPL on the local processor does not prevent other processors in the system from reading or modifying a shared database. Unless other steps are taken to notify the other processors that the database is "owned," such contention could potentially result in corrupted data and system failures.

A **spin lock** is a semaphore associated with a set of system structures, fields, or registers whose integrity is critical to the performance of a specific operating system task. The scheduler and the memory management subsystem thus have their own spin locks, as does each fork processing level and each device controller. Because a spin lock can be owned by only one processor in the system at a time, other processors

attempting to acquire the same spin lock are prevented from reading from or writing into the database it protects. Refer to the data structures contained in *VMS Device Support Reference Manual* for an illustration of the spin lock (SPL) structure and a description of its contents.

There are two categories of spin lock:

- The structure of a **static spin lock** is permanently assembled into the system. As a result, its existence and definition are fixed from one system to another. Static spin locks are accessed as indexes into a vector of longword addresses called the **spin lock vector** and pointed to by SMP$AR_SPNLKVEC. The system spin locks and fork locks listed in Table 3–3 are static spin locks.
- A **dynamic spin lock** is a spin lock that is created based on the I/O configuration of a particular system. One such dynamic spin lock is the device lock SYSGEN creates when configuring a particular device. This device lock synchronizes access to the device's registers and certain unit control block (UCB) fields. VMS creates a dynamic spin lock by allocating space from nonpaged pool, rather than by assembling the lock into the system as it does in the creation of a static spin lock. Section 3.2.2 describes device locks.

Table 3–3 lists, in order of increasing logical rank, the static spin locks. For each system spin lock or fork lock, the table records its index into the spin lock vector, its synchronization IPL, and a brief description of its function.

Table 3–3 Static Spin Locks

Lock Name	Lock Index	Synchronization IPL	Description
QUEUEAST	SPL$C_QUEUEAST	6 (IPL$_QUEUEAST)	Fork lock for executing a fork process at IPL 6
FILSYS	SPL$C_FILSYS	8 (IPL$_FILSYS)[1]	Lock on file system structures
IOLOCK8[2] SCS[2]	SPL$C_IOLOCK8 SPL$C_SCS	8 (IPL$_IOLOCK8 IPL$_SCS) [1]	Fork lock for executing a fork process at IPL 8
PR_LK8	SPL$C_PR_LK8	8 (IPL$_IOLOCK8)[1]	Primary CPU's private lock for IPL 8
TIMER	SPL$C_TIMER	8 (IPL$_TIMER)[1]	Lock for adding and deleting timer queue entries and searching the timer queue[3]
JIB	SPL$C_JIB	8 (IPL$_JIB)[1]	Lock for manipulating job nonpaged pool quotas as reflected by the fields JIB$L_BYTCNT and JIB$L_BYTLM in the job information block

[1] IPL$_TIMER, IPL$_SCHED, IPL$_SCS, IPL$_JIB, IPL$_MMG, IPL$_FILSYS, and IPL$_IOLOCK8 are all synonyms for IPL$_SYNCH.

[2] These names refer to the same spin lock.

[3] The HWCLK spin lock implicitly locks the timer queue element at the head of the timer queue by locking the quadword representing its due time (EXE$GQ_1ST_TIME).

(continued on next page)

Table 3–3 (Cont.) Static Spin Locks

Lock Name	Lock Index	Synchronization IPL	Description
MMG	SPL$C_MMG	8 (IPL$_MMG)[1]	Lock on VMS memory management, PFN database, swapper, modified page writer, and creation of per-CPU database structures
SCHED	SPL$C_SCHED	8 (IPL$_SCHED)[1]	Lock on process control blocks, scheduler database, and mutex acquisition and release structures
IOLOCK9	SPL$C_IOLOCK9	9 (IPL$_IOLOCK9)	Fork lock for executing a fork process at IPL 9
PR_LK9	SPL$C_PR_LK9	9 (IPL$_IOLOCK9)	Primary CPU's private lock for IPL 9
IOLOCK10	SPL$C_IOLOCK10	10 (IPL$_IOLOCK10)	Fork lock for executing a fork process at IPL 10
PR_LK10	SPL$C_PR_LK10	10 (IPL$_IOLOCK10)	Primary CPU's private lock for IPL 10
IOLOCK11	SPL$C_IOLOCK11	11 (IPL$_IOLOCK11)	Fork lock for executing a fork process at IPL 11
PR_LK11	SPL$C_PR_LK11	11 (IPL$_IOLOCK11)	Primary CPU's private lock for IPL 11
MAILBOX	SPL$C_MAILBOX	11 (IPL$_MAILBOX)	Lock for sending messages to mailboxes
POOL	SPL$C_POOL	11 (IPL$_POOL)	Lock on nonpaged pool database
PERFMON	SPL$C_PERFMON	15 (IPL$_PERFMON)	Lock for I/O performance monitoring
INVALIDATE	SPL$C_INVALIDATE	IPL$_INVALIDATE[4]	Lock system space translation buffer (TB) invalidation
VIRTCONS	SPL$C_VIRTCONS	IPL$_VIRTCONS[5]	Lock for ownership of the virtual console
HWCLK	SPL$C_HWCLK	22 or 24	Lock on interval clock database, including the quadword containing the due time of the first timer queue element and the quadword containing the system time
MEGA	SPL$C_MEGA	31 (IPL$_MEGA)	Lock for serializing access to fork and wait queue
MCHECK[2] EMB[2]	SPL$C_MCHECK SPL$_EMB	31 (IPL$_MCHECK IPL$_EMB)	Lock for synchronizing certain machine error handling and for allocating and releasing error logging buffers

[1]IPL$_TIMER, IPL$_SCHED, IPL$_SCS, IPL$_JIB, IPL$_MMG, IPL$_FILSYS, and IPL$_IOLOCK8 are all synonyms for IPL$_SYNCH.

[2]These names refer to the same spin lock.

[4]The IPL associated with this spin lock is determined at system initialization and is one less than that of the system's interprocessor interrupt. On VAX 88*x*0 and VAX 83*x*0 systems, the value is 19. On VAX 6000-series and VAX 9000-series systems, the value is 21.

[5]The IPL associated with this spin lock is determined at system initialization and is the level of the system's interprocessor interrupt. On VAX 6000-series and VAX 9000-series systems, the value is 22; on other VAX multiprocessing systems, the value is 20.

Drivers rarely need to obtain system spin locks or fork locks explicitly; the VMS routines that initiate driver processing and access resources protected by a spin lock generally obtain and release these locks as required. However, a driver must obtain the appropriate device locks

whenever it must access data synchronized at device IPL; for instance, in its interrupt service routine.

VMS provides a set of macros, listed in Table 3–2 and described in full in the macro chapter of the *VMS Device Support Reference Manual*, that call the system's spin lock acquisition and releasing routines.

Three factors control the successful acquisition of a spin lock: IPL, rank, and ownership.

IPL

The processor must be executing at an IPL equal to or below the spin lock's synchronization IPL (SPL$B_IPL). In keeping with the rules discussed in Section 3.1.4.2, a processor should not lower the IPL of its thread of execution in the process of acquiring a spin lock. Thus, in acquiring a spin lock, a processor may or may not raise its IPL, depending upon whether it is executing already at the spin lock synchronization IPL. VMS supplies spin lock acquisition macros (DEVICELOCK, FORKLOCK, and LOCK) that, in calling appropriate VMS routines, raise IPL automatically in the course of obtaining the requested spin lock. Once it owns the spin lock, the processor can raise its IPL above the IPL at which the spin lock was acquired, but it should not lower it below that level.

Rank

A processor can own multiple spin locks simultaneously, but must obtain these spin locks in increasing order of rank. (Table 3–3 lists the spin locks in order of rank.) In other words, a processor that owns one or more spin locks should not attempt to acquire a spin lock whose logical rank[4] is less than a spin lock it already holds. It does not need to acquire all spin locks of intervening rank. This rule is meant to avoid potential deadlocks in the acquisition of system spin locks and fork locks, and does not pertain to device locks. The processor may release spin locks in any order, as long as any attempt to reacquire those spin locks acquires them in ascending order.

Note that the concept of rank is independent of IPL. At any given synchronization IPL, there may be many spin locks, each of which is ranked according to its position in Table 3–3.

Ownership

The spin lock must not be owned by any other processor. If the spin lock is currently owned by another processor, a requesting processor **spin waits** for the lock to become available. That is, it executes in a loop, waiting for the processor that owns the spin lock to release it. If a spin lock is owned, its owner field (SPL$L_OWN_CPU) contains an identifier that indicates which processor in the multiprocessor system owns the spin lock.

It is legal for a processor to nest acquisitions of a given spin lock. In other words, if a processor attempts to acquire a spin lock that it currently owns, the acquisition will succeed. VMS provides a mechanism whereby such a processor can release a single acquisition or all acquisitions of a spin lock.

[4] The physical rank of a spin lock is the inverse of its logical rank. See the description of the SPL$B_RANK field in the *VMS Device Support Reference Manual* for additional information.

3.2.1 Fork Locks

In its simplest form, a fork lock is a static spin lock that synchronizes the right of a fork process to execute at a specified IPL in a VMS multiprocessing system. Fork locks exist for each of the fork IPLs from IPL 8 to 11. A driver indicates the fork lock under which it processes, and by implication its fork IPL, by specifying a fork lock index in its driver prologue table (using the DPT_STORE macro as described in Section 6.1).

Those code threads that must synchronize with another fork thread use the same fork lock. For instance, the fork processes of drivers whose devices share the resources of a common adapter *must* synchronize themselves by means of a common fork lock. These code threads fork not necessarily to lower IPL, but rather to wait for the availability of a common resource such as a controller data channel or map registers (see Section 3.4). The VMS routines that acquire and release these resources ensure that the fork lock is acquired and released as necessary.

Drivers rarely need to obtain a fork lock explicitly. VMS places the driver fork process into execution (originally by EXE$INSIOQ and, by implication, by IOC$REQCOM) at fork IPL holding the appropriate fork lock. In addition, the fork dispatcher obtains the fork lock associated with the driver fork process before it restores its context and resumes its execution. (Section 3.3.3 describes these concepts in greater detail.)

Note that, if a driver fork process is not placed into execution by one of these means, it must itself expressly obtain the fork lock.

As an example, consider a driver fork process activated by a timer wakeup associated with a timer queue element (TQE) previously queued by the driver. The software timer interrupt service routine does raise IPL to IPL 8 (IPL$_SYNCH) and obtain certain spin locks prior to dequeuing the TQE and placing it into execution, but it does *not* obtain the driver's fork lock. Thus, even though the driver's fork IPL may be IPL$_SYNCH, the driver will not be properly synchronized at fork level unless it first obtains the appropriate fork lock.

3.2.2 Device Locks

A device lock represents a lock on an individual adapter or controller. A processor executing a code thread that accesses a device's registers or certain fields in its unit control block (UCB) that reflect its status does so while holding the corresponding device lock.

UCBs are protected by a device lock common to all units on the same adapter or common to the entire system, depending upon the type of device. A device lock is dynamically created by the System Generation Utility (SYSGEN) when it creates a channel request block (CRB). SYSGEN stores the address of the device lock in the CRB (CRB$L_DLCK) and later copies it to the unit control block (UCB$L_DLCK) as a UCB is created for each unit on the controller.

The acquisition of device locks is exempt from the spin lock rank rule. As long as the processor does not violate IPL synchronization, it may successfully obtain an unowned device lock while holding any system spin lock and, likewise, may successfully obtain unowned system spin locks while holding a device lock. However, a processor can acquire only one device lock at a given IPL.

3.3 Device Driver Synchronization

This section describes how VMS and driver processing maintain synchronization during the processing of a general I/O request. It later focuses on the specific strategies drivers employ to synchronize at the device and fork levels.

3.3.1 Overview of the Synchronization of an I/O Operation

Figure 3–1 diagrams the general flow of the processing of a single I/O request, as synchronization is achieved by raising and lowering IPL, and, in a multiprocessing environment, by also obtaining and releasing the necessary spin locks.

Figure 3–1 Synchronizing I/O Request Processing

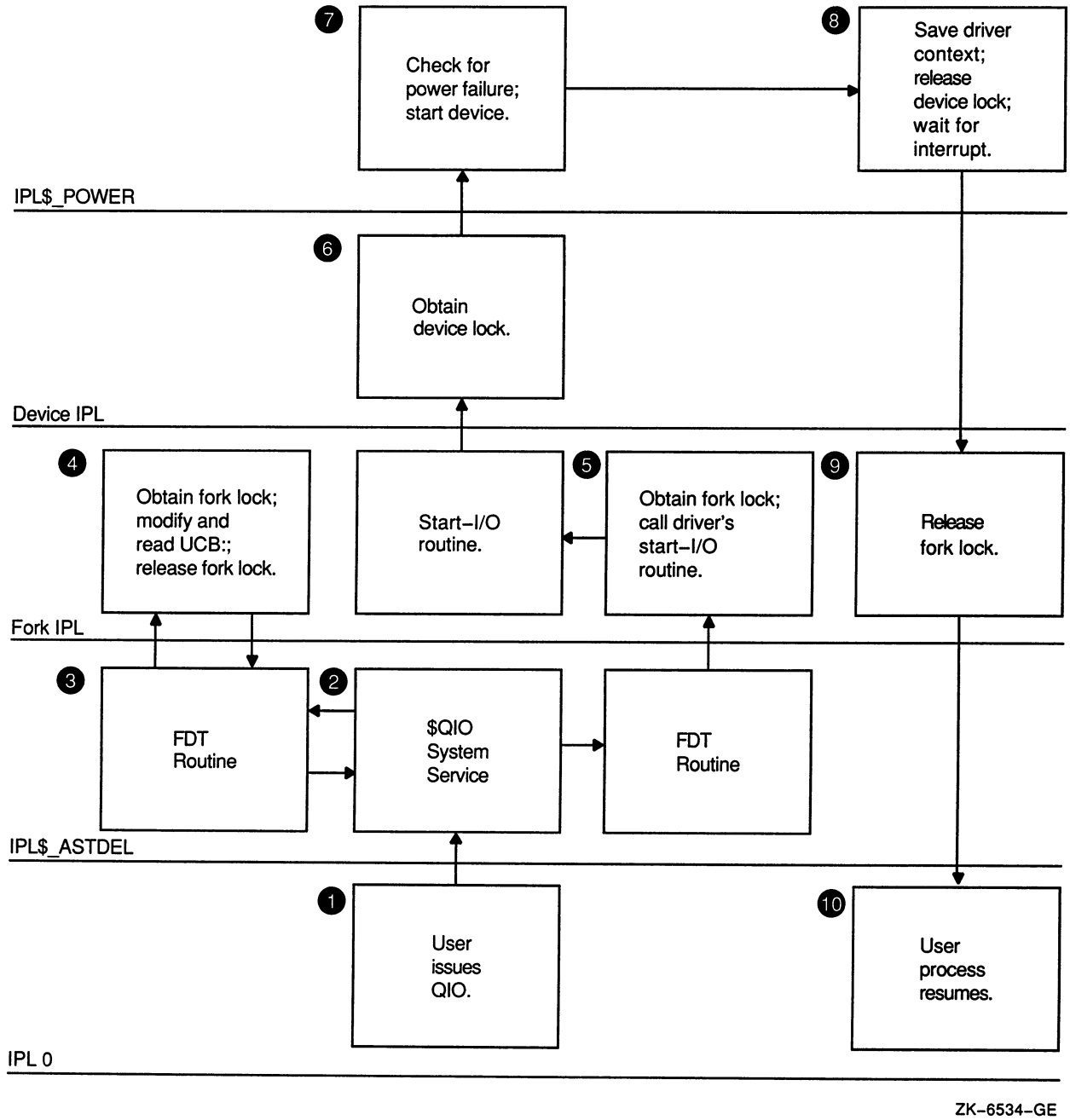


Figure 3–1 illustrates the following events:

❶ The user program, executing at IPL 0, issues a $QIO system service call.

❷ The $QIO system service raises IPL to IPL$_ASTDEL to prepare the I/O request according to the arguments included in the call.

❸ The driver's FDT routines execute, mainly at IPL$_ASTDEL.

Note that during IPL 0 processing and FDT routine activity at IPL$_ASTDEL, the process requesting the I/O is susceptible to being rescheduled. In a multiprocessing environment, such an event could cause I/O processing to resume on a different processor from that on which it was started.

❹ In certain rare circumstances, an FDT routine must read or modify the device's UCB. Because most fields in the UCB may be shared by fork processes running systemwide it is important that, if the FDT routine must use them, it issue the FORKLOCK macro to obtain the appropriate fork lock and raise to fork IPL. (When finished, it relinquishes this synchronization by issuing the FORKUNLOCK macro.)

❺ The continuation of VMS preprocessing of the I/O request—or the completion of a previous I/O request on the device unit—ensures that the driver's start-I/O routine is placed into execution at fork IPL and, in a multiprocessing system, holding the corresponding fork lock. The start-I/O routine accesses various UCB fields and contends for adapter resources synchronized systemwide by the fork lock.

❻ Once it has further prepared the I/O request and obtained the required resources, it generally must access device registers. Device registers and those UCB fields that record their status are synchronized at device IPL. A processor in a VMS multiprocessing system must hold the appropriate device lock to access the device database.

❼ While executing a critical code sequence, such as those instructions that start a device, the start-I/O routine raises the IPL of the local processor to IPL$_POWER to check for a processor power failure. In a VMS multiprocessing environment, the executing processor retains the device lock during this sequence.

❽ After it activates the device, the start-I/O routine calls a VMS routine that saves the driver's context in the UCB fork block, suspends driver processing, releases the device lock, if held, and restores IPL to a previous level.

❾ VMS at this point returns control to the code that initiated the fork thread where, in a VMS multiprocessing system, the fork lock is released.

❿ After VMS services interrupts at intervening IPLs, the user process resumes.

Figure 3–2 illustrates the synchronization involved in the completion of an I/O request from the point of the device interrupt to the delivery of ASTs to the user program. There is little linear flow involved in the completion of an I/O request. The servicing of interrupts, represented by jagged lines in the figure, the requesting of software interrupts, and the REI instruction contribute to the flow that completes an I/O request.

Figure 3–2 Synchronizing I/O Request Completion

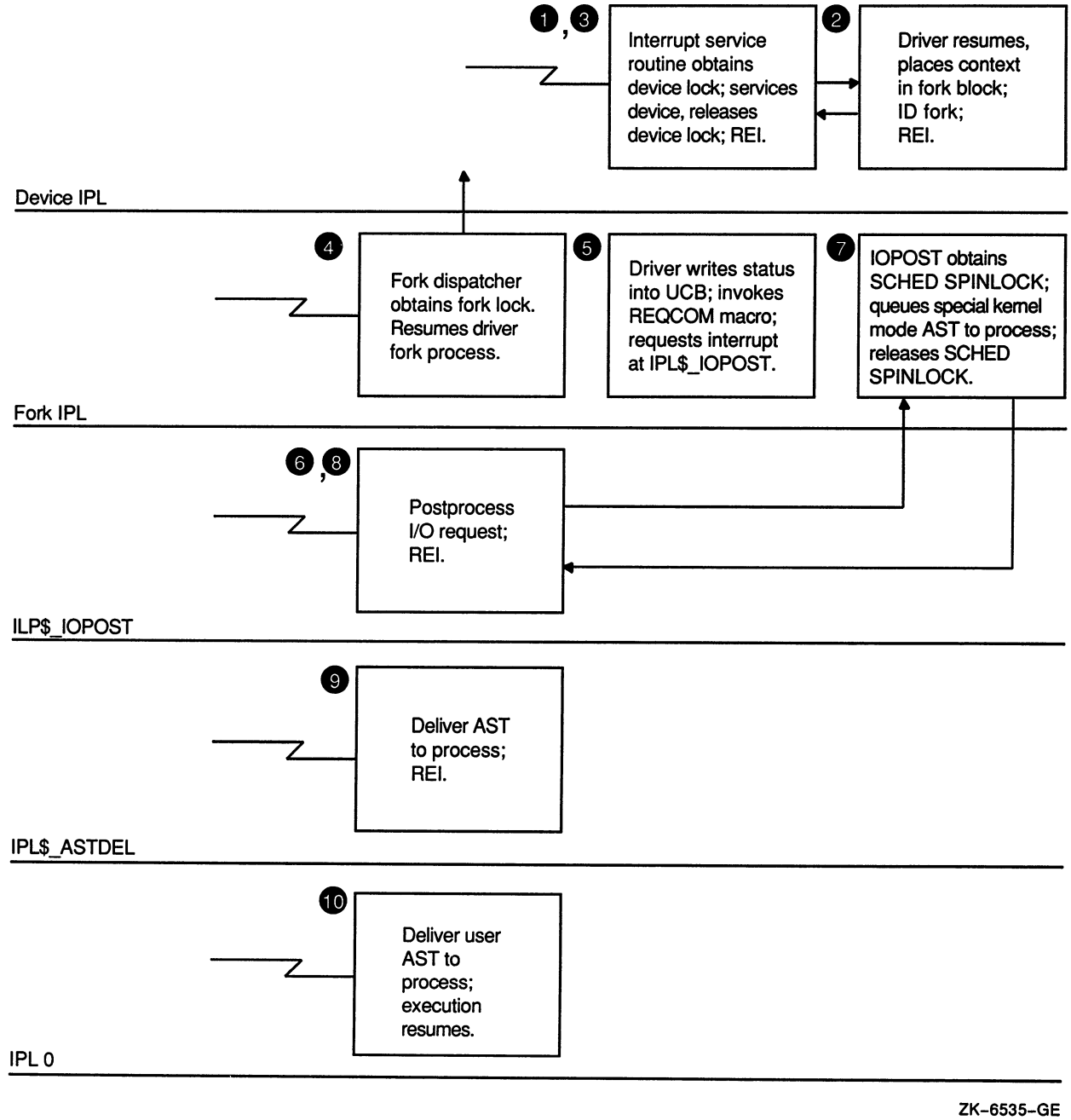


Figure 3–2 illustrates the following events:

❶ A device interrupt in the range of IPL 20 to IPL 23 triggers the execution of the driver's interrupt service routine. The interrupt service routine locates the device unit's UCB and, in a VMS multiprocessing system, immediately obtains the appropriate device lock. After it analyzes the interrupt and determines that it is expected,

it reactivates the driver, still at device IPL and holding any acquired device lock.

❷ The driver briefly examines and/or saves the contents of the device's registers, but, in order to permit other device interrupts to be serviced and to allow other high priority system tasks to proceed, it lowers its own priority. The driver accomplishes this by requesting VMS to save some driver context in the UCB fork block and place it into one of the processor-specific fork queues at IPLs 8 to 11 serviced by the fork dispatcher. When it does so, VMS returns control to the driver's interrupt service routine.

❸ The interrupt service routine releases any acquired device lock and issues an REI instruction to dismiss the device interrupt.

❹ When IPL drops below the driver's fork IPL, the fork dispatcher restores the context of the driver and resumes its execution. In a VMS multiprocessing system, the fork dispatcher obtains the necessary fork lock prior to placing the driver into execution.

❺ Still synchronized at fork level, the driver fork process analyzes the success of the I/O operation and writes status into R0 and R1. VMS then inserts the IRP in the systemwide I/O postprocessing queue, requests a software interrupt at IPL$_IOPOST, and starts any I/O request that may be waiting for the device. Eventually, VMS returns to the fork dispatcher and, if no other fork processes are queued for that IPL, issues an REI instruction to dismiss the software interrupt.

❻ When the processor's IPL falls below IPL$_IOPOST, the I/O postprocessing routine removes the IRP from the I/O postprocessing queue, adjusts process quota usage, and deallocates system buffers for write functions.

❼ When the routine finishes processing the IRP, it queues a special kernel-mode AST to the process that issued the original $QIO request. To accomplish this, it obtains the SCHED spin lock (raising to IPL$_SYNCH in the process) and calls another VMS routine that queues the AST to the process's PCB. It then releases the SCHED spin lock.

❽ The I/O postprocessing routine continues execution at IPL$_IOPOST until it has serviced all entries in the postprocessing queue. It then issues an REI instruction to dismiss this software interrupt.

❾ The special kernel-mode AST routine executes at IPL$_ASTDEL. It completes the transfer of the results and status of the I/O request to the user process.

❿ The special kernel-mode AST routine can queue a user-mode AST routine to the user process. When the user process has been rescheduled and its context reloaded, the user-mode AST routine executes at IPL 0.

3.3.2 Synchronizing the Device Database

A **device database** ordinarily consists of the device or adapter registers, plus some storage in the UCB (or in another data structure) that reflects the status of the device. Routines that access data in the device database must do so at device IPL (UCB$B_DIPL) in order to maintain synchronization. Generally, only three driver routines contend for access to the device database.

- Interrupt service routine
- Start-I/O routine when loading or reading device registers
- Timeout handling routine

In a VMS uniprocessing environment, the start-I/O routine raises its IPL to device IPL using the DSBINT macro. VMS calls the driver's timeout handling routine at device IPL. Because the interrupt dispatcher invokes it at device IPL, the driver's interrupt service routine does not need to acquire additional synchronization.

In a VMS multiprocessing environment, these routines must also hold the appropriate device lock (UCB$L_DLCK). The device lock protecting the device database is a dynamic spin lock, created by SYSGEN when the device is configured and its channel request block (CRB) is created. The address of the device lock is first stored in CRB$L_DLCK and is moved to UCB$L_DLCK as corresponding UCBs are allocated for each unit on the controller. VMS calls the driver's timeout handling routine at device IPL with the device lock held. The start-I/O routine and the interrupt service routine must explicitly obtain such synchronization by invoking the DEVICELOCK macro.

The start-I/O routine and timeout handling routine are additionally synchronized at driver fork level. VMS raises IPL to fork level and obtains the corresponding fork lock before transferring control to them. This is not the case, however, with a driver's interrupt service routine. A device's interrupt service routine usually does not hold the fork lock. However, it may have preempted a thread holding the fork lock, or a fork thread may be running in parallel on another processor. Therefore, an interrupt service routine must not change any fields in the UCB that are protected by the fork lock. To access these fields, an interrupt service routine must fork, as described in Section 3.3.3.1.

3.3.3 Synchronizing at Driver Fork Level

A large part of driver code executes in the context of a **fork process**. As a fork process, driver code that must access data in its fork database does so at a single, specific fork IPL (from IPL 8 to IPL 11) and—in a VMS multiprocessing environment—holding a single, specific fork lock (see Section 3.2.1). The **fork database** consists of those fields in the unit control block (UCB) not explicitly synchronized at device level and such adapter or controller resources as map registers or data paths.

The system routine EXE$INSIOQ initially creates a driver fork process as it attempts to deliver a preprocessed I/O request to the driver's start-I/O routine. If the device unit is busy (that is, a fork process is already active servicing a prior request for that device), EXE$INSIOQ inserts the IRP into the UCB's pending-I/O queue. If the device unit is not busy, EXE$INSIOQ calls IOC$INITIATE to transfer control to the driver's start-I/O routine. The start-I/O routine begins to execute at fork IPL holding the associated fork lock, if necessary.

When the driver fork process later calls IOC$REQCOM to complete processing of a prior I/O request, IOC$REQCOM executes within the driver fork process, dequeues the next IRP on the pending-I/O queue, and begins processing it.

Like other processes, fork processes can be interrupted or suspended. The local processor interrupts a fork process when the processor receives a request for an interrupt at a higher priority level. To minimize the number of interruptions, fork processes sometimes execute at raised IPLs, and even raise their IPL to block all other interrupts, if necessary.

VMS stalls a driver's fork process when the process requests an unavailable resource such as a controller's data path (see Section 3.4). When suspended, a driver fork process, like other processes, preserves some context information. As VMS preserves some of the context of a normal process in its hardware PCB, so it preserves a driver fork process's context—however abbreviated—in a fork block. **Fork context** consists of the following:

- Two general purpose registers (R3 and R4)
- The program counter (PC)
- A fork block (usually the UCB), the address of which is in R5 at the time of the suspension

Minimal context helps ensure that, when a driver fork process is ready to be resumed, the resulting context-switching occurs swiftly.

3.3.3.1 Forking and the VMS Fork Dispatcher

Forking allows high IPL code to do the following:

- Continue executing a particular code thread at a lower IPL than the IPL at which the code thread was initiated
- Synchronize with other code executing at the lower IPL

Usually, a driver forks after servicing a device interrupt at an IPL from 20 to 23. By forking, the driver lowers the IPL at which it continues to process the device interrupt from device IPL to fork IPL (8 to 11). Forking not only allows the driver to process efficiently that part of interrupt request processing that is not time critical, but it allows the driver to synchronize its execution with other fork process code threads initiating I/O. For example, forking helps the driver synchronize its use of a device unit's UCB with other code threads interested in the structure. Moreover, the driver, by forking after completing the initial servicing of a device interrupt, allows other device interrupts to occur at that device IPL.

To fork, either the driver's interrupt service routine or the start-I/O routine, when resumed by the interrupt service routine, invokes the VMS macro IOFORK. The IOFORK macro saves fork process context in the UCB fork block, places the fork block in the local processor's fork queue for the specific fork IPL, and requests a software interrupt for that IPL. When that interrupt is ultimately serviced, driver fork processing resumes at the lower level.

There are other specialized instances in which a device driver may fork. As discussed in Section 11.1.5, the driver's unit initialization routine or controller initialization routine, while executing at IPL 31, may fork in order to permanently allocate controller resources, system nonpaged dynamic memory, or system page-table entries. To fork, these routines use the VMS macro FORK. The FORK macro allows a driver to fork, utilizing the fork block, the address of which is placed in R5. Because the channel request block (CRB) is available to these routines and contains a fork block, they invoke the VMS macro FORK with the address of the CRB in R5.

One interrupt service routine (EXE$FORKDSPTH) handles all fork-process dispatching on each processor in a VAX system. When the processor grants an interrupt at fork IPL, the fork dispatcher saves R0 to R5 on the stack and processes the local fork queue that corresponds to the IPL of the interrupt. To do so, it removes an entry from the fork queue, restores the fork process context from the fork block, obtains ownership of the fork lock specified in the fork block (in a VMS multiprocessing system), and reactivates the suspended fork process.

When that fork process is completed, the dispatcher releases the fork lock and examines the fork queue. If an entry exists on the queue, the fork dispatcher removes it, restores the context of the fork process, secures the fork lock specified in the fork block, and reactivates the fork process. This sequence is repeated until the fork queue is empty. When the queue is empty, the fork dispatcher restores R0 to R5 from the stack and dismisses the interrupt with an REI instruction.

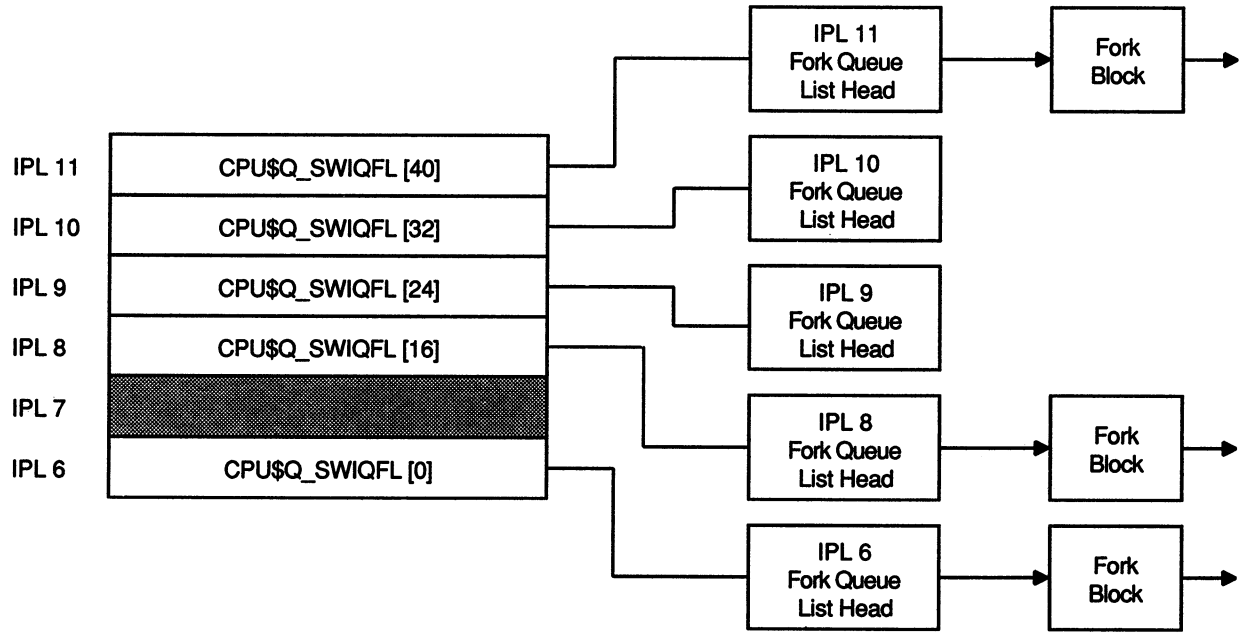
Figure 3–3 illustrates the fork queue structure.

3.3.3.2 Restrictions on Fork Processes

A driver fork process executes under the following constraints:

- It should not attempt to refer to the address space of the process initiating the I/O request.
- It can use only R0 to R5 freely; it must save other registers before use and restore them after use. Use of registers other than R0 to R5 is strongly discouraged.
- It must clean up the stack after use; the stack must be in its original state when the fork process relinquishes control to any VMS routine.
- It must execute at IPLs between the driver's fork IPL and IPL$_POWER. It must not lower IPL below the driver's fork IPL except by creating a fork process to execute at a lower IPL.

Figure 3–3 Processor-Specific Fork Queue Structure



- If executing in a VMS multiprocessing environment, it cannot attempt to obtain system spin locks with lower ranks than that of its fork lock.
- When it returns control to the fork dispatcher, the fork process must be at the same fork IPL and, if executing on a VMS multiprocessing system, own the appropriate fork lock.

3.4 Resource Wait Queues

The processing of an I/O request often requires shared system resources such as memory and I/O adapter map registers. Drivers that depend on such resources synchronize access to these resources and their respective resource wait queues by executing at fork IPL and, in a VMS multiprocessing environment, obtaining ownership of the associated fork lock.

The $QIO system service and fork processes call VMS routines to allocate and deallocate shared system resources. Because the resources are limited, I/O processing might be delayed until any such needed resources are released. Thus, synchronization of access to these resources can have a substantial impact on the processing of I/O requests.

For example, the $QIO system service calls a VMS routine to allocate nonpaged system space for an IRP. If there is insufficient nonpaged pool, the routine calls another VMS routine to save the process context and change the process state to resource-wait mode (also called miscellaneous wait. or MWAIT). As a result of waiting, the process is a candidate to

be swapped out of memory. When nonpaged pool becomes available, the scheduler reschedules the process.

During fork process execution at elevated IPLs, driver context is very small. At any point, the driver can obtain all details about an I/O request by referring to the I/O database (see the data structures in *VMS Device Support Reference Manual*). The driver needs only the address of the device's UCB, which is the key to the rest of the database. Therefore, VMS routines that control driver resources, such as map registers, use fork blocks and resource-wait queues to save minimal driver context. Each entry in a queue is a fork block (or UCB) that contains R3, R4, and the continuation PC of the waiting fork process.

When the awaited resource becomes available, the routine controlling the resource performs the following steps:

- Restores the UCB address to R5
- Restores the saved registers R3 and R4
- Grants the resource
- Transfers control to the saved driver return PC address

Because the VMS routine that controls a particular resource stalls any driver that requests an unavailable resource, drivers are unaware of execution being suspended and subsequently reactivated. *Drivers must not leave anything on the stack, or in general purpose registers, other than R3, R4, and R5, when calling a routine that might suspend the driver's execution.*

3.4.1 Competing for a Controller's Data Channel

A controller's data channel is a VMS synchronization mechanism that guarantees that only one unit of a multiunit controller uses the controller at one time.

Devices that share a controller, such as disk units, own the controller's data channel only when a VMS routine assigns the channel to the unit's fork process. The device driver's start-I/O routine issues the REQPCHAN macro to obtain the channel.

In contrast, a device unit on a single-unit controller always owns the controller's data channel. The device driver's controller (or unit) initialization routine affirms this fact by moving the address of the device's UCB into IDB$L_OWNER. Generally, the driver's start-I/O routine does not request a single-unit controller.

In each case, the driver's start-I/O routine must take steps to synchronize its access to device registers with any access of these registers by the driver's interrupt service routine. The routine does so by issuing the DEVICELOCK macro (as described in Section 3.1.4). The DEVICELOCK macro raises IPL to device IPL and, in a VMS multiprocessing system, obtains the device lock associated with the controller.

An RK611 controller, for example, controls as many as eight RK06/RK07 devices. The disk driver's fork process must gain control of the controller's data channel before starting an I/O operation on the unit associated with the fork process. The disk driver's start-I/O routine uses the following sequence to start a seek operation on an RK07 device:

1 The start-I/O routine requests the controller's data channel by invoking a VMS channel arbitration macro (REQPCHAN).

2 The VMS routine tests the CRB mask field to determine whether the controller's data channel is available.

3 If the channel is available, the VMS routine allocates the channel to the fork process and returns the address of the device's CSR to the fork process.

  If the channel is busy, the VMS routine saves the driver fork context in the UCB fork block and inserts the fork block address in the controller's channel wait queue.

4 When the fork process resumes execution, the process owns the controller channel. The fork process can then obtain the device lock (raising IPL to device IPL) and modify the device's registers to activate the device.

5 The driver's start-I/O routine then requests the VMS operating system to suspend driver processing in anticipation of an interrupt or timeout and to release the channel.

6 The VMS channel-releasing routine assigns channel ownership to the next fork process in the channel wait queue, loads the CSR address into a general register, and reactivates the suspended fork process.

7 The reactivated fork process continues execution as though the channel had been available in the first place.

The VMS channel-arbitration routines keep track of controller availability using a flag field in the CRB. The fork process must always request and release the controller's data channel by invoking these routines.

4 Overview of I/O Processing

Under the VMS operating system, I/O processing occurs in three major phases:

- I/O request preprocessing
- Device activation and subsequent handling of the device interrupt
- I/O postprocessing

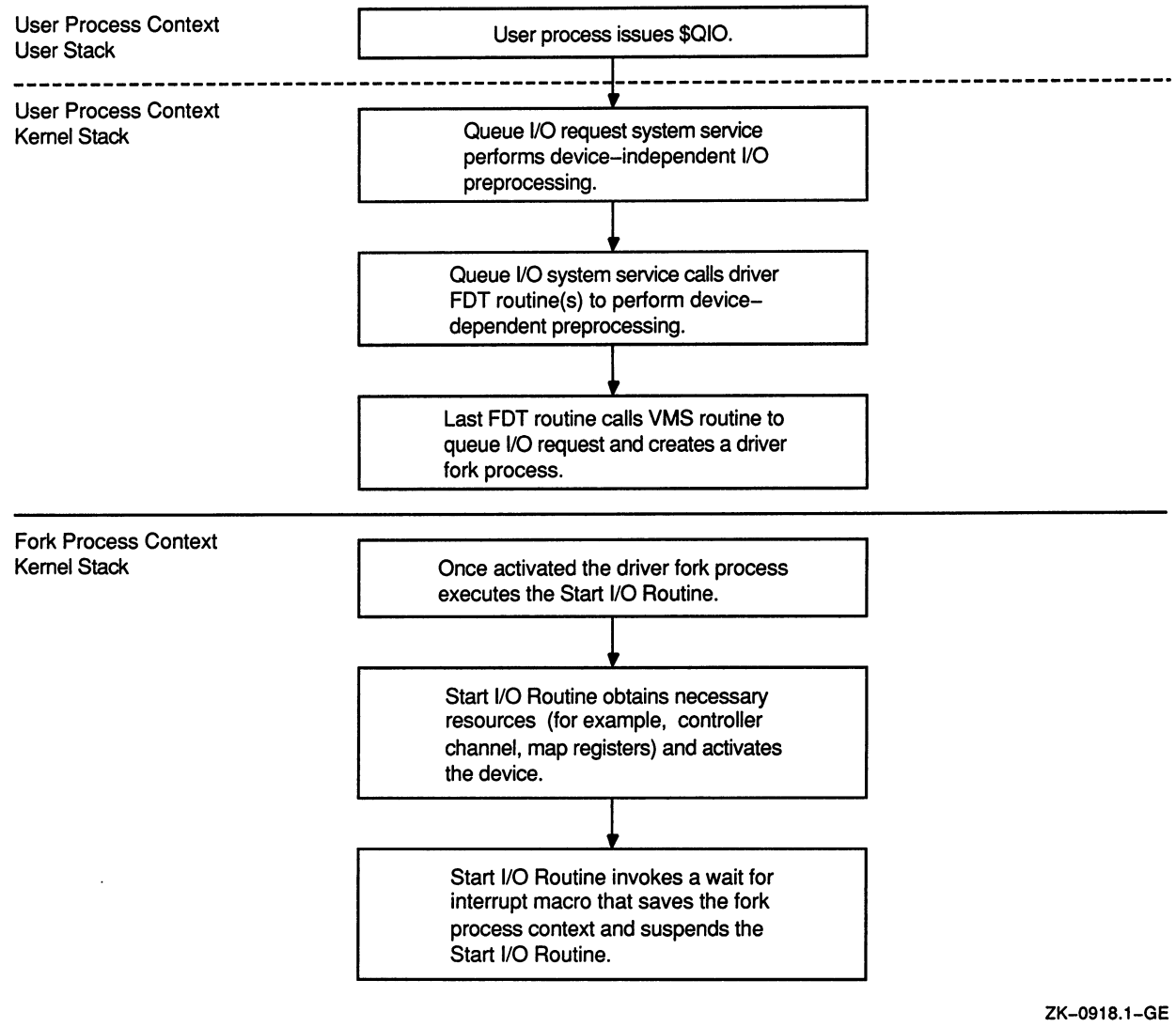
When a user process issues an I/O request, the Queue I/O Request ($QIO) system service gains control and coordinates preprocessing of the request. The last driver FDT routine called by the $QIO system service calls a VMS routine that creates a driver fork process to execute the driver's start-I/O routine. This routine activates the device.

When the transfer is completed, the device requests an interrupt that results in execution of the driver's interrupt service routine. This routine handles the interrupt and requests resumption of the driver fork process to perform device-dependent I/O postprocessing. The driver fork process finally transfers control to the system to perform device-independent I/O postprocessing. Figure 4–1 illustrates the sequence of events.

The $QIO system service is dispatched by means of a corresponding system service vector in process P1 space. This vector contains a CHMK instruction that causes an exception that alters the process's access mode to kernel and dispatches to the service-specific procedure, EXE$QIO.

For the purposes of the discussion in this section, as well as the rest of the book, Figure 4–2 portrays the flow of an I/O request from its system service entry point to its servicing by VMS executive routines and driver code. Discussion of other entry points appears in Chapters 8, 9, and 10.

Figure 4–1 Sequence of Driver Execution



ZK–0918.1–GE

(continued on next page)

Figure 4–1 (Cont.) Sequence of Driver Execution

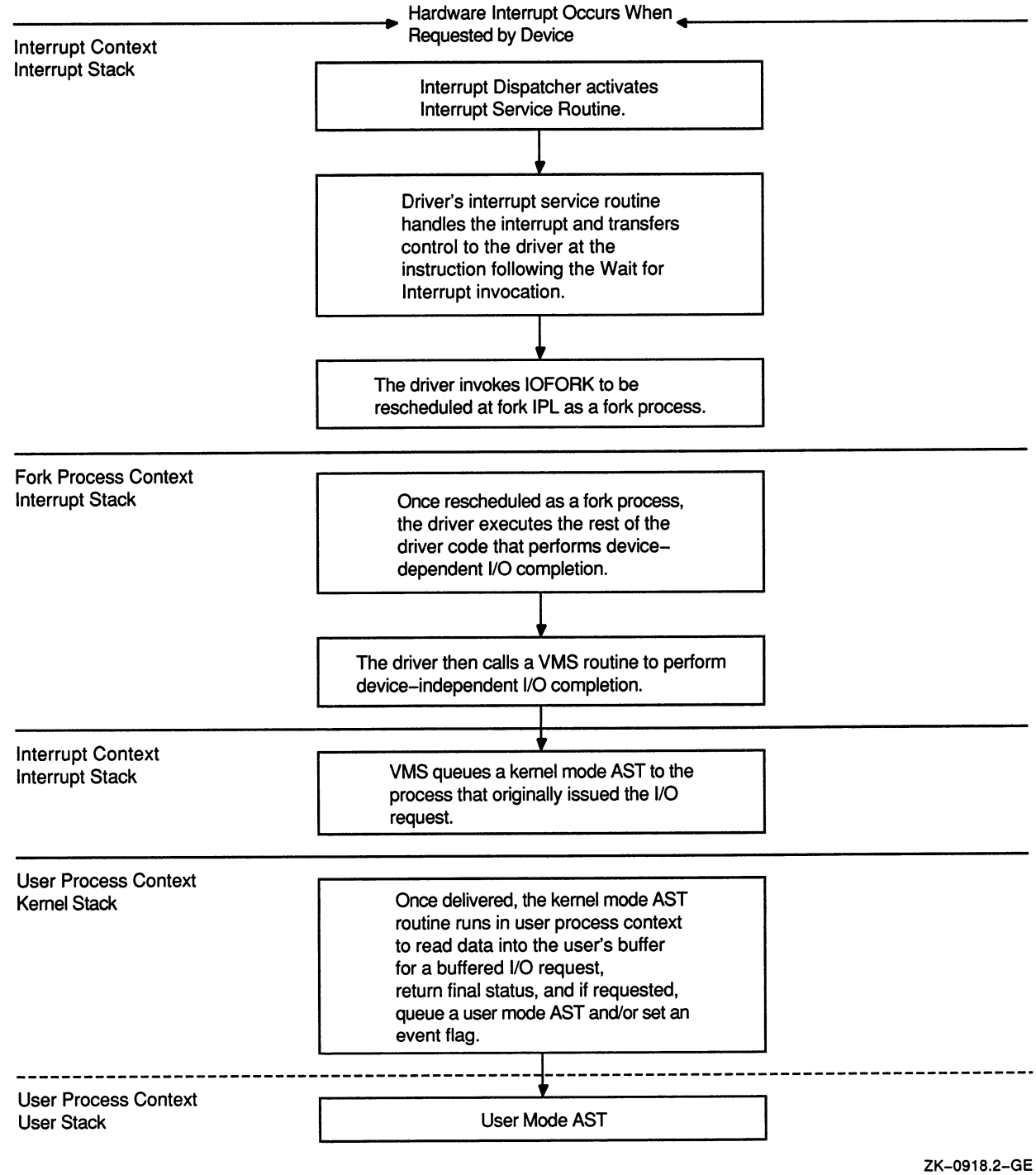
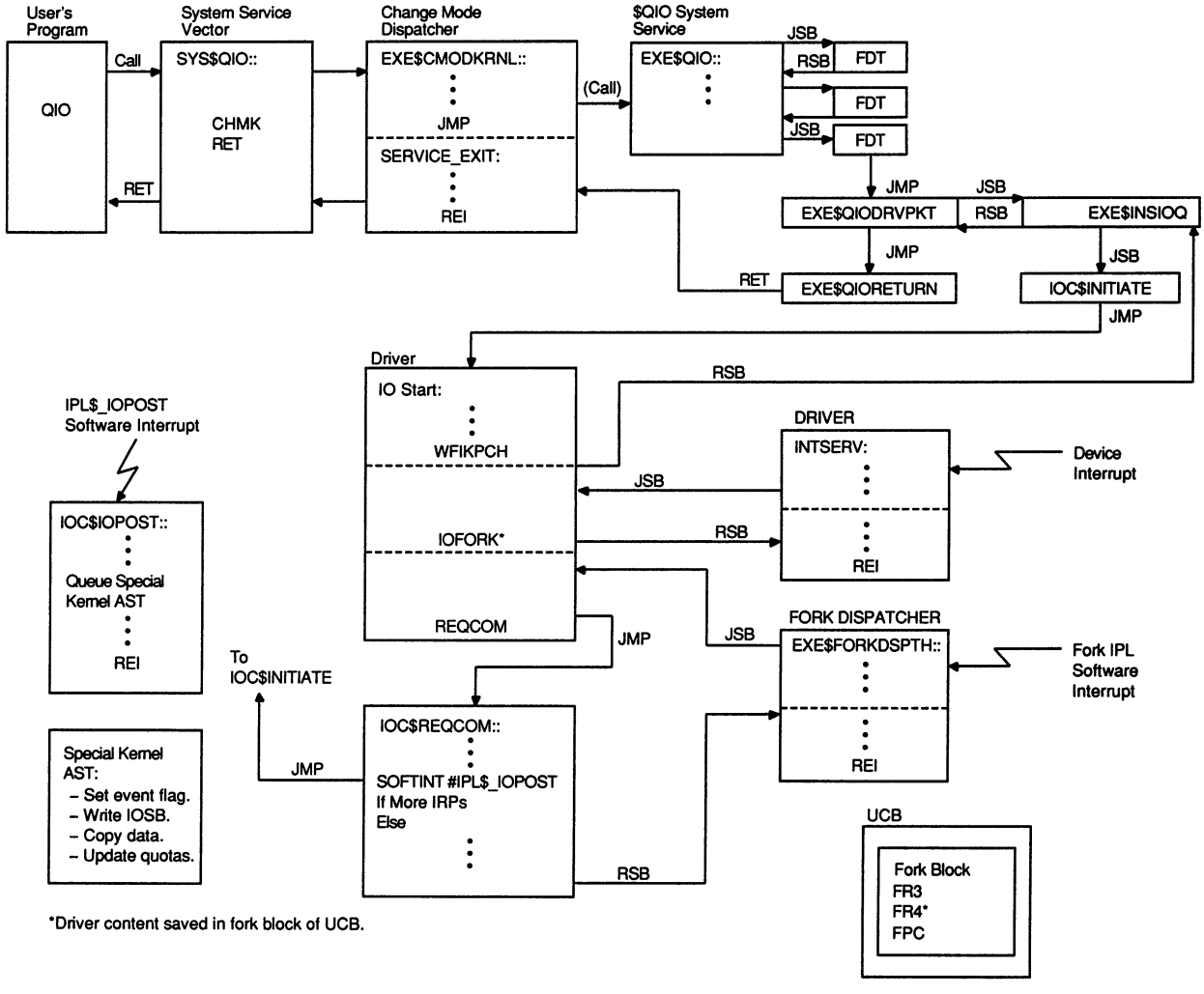


Figure 4–2 Detailed Sequence of VMS I/O Processing



4.1 Preprocessing an I/O Request

EXE$QIO performs device-independent preprocessing of an I/O request and calls driver FDT routines to perform device-dependent preprocessing. To preprocess an I/O request, EXE$QIO takes the following steps:

- Verifies that the requesting process has assigned a process I/O channel to the target device
- Locates the device driver in the I/O database
- Validates the I/O function code
- Checks process I/O request quotas
- Validates the I/O status block

- Allocates and sets up the I/O request packet (IRP)
- Calls driver FDT routines to perform device-dependent preprocessing

4.1.1 Process I/O Channel Assignment

The first step in preprocessing an I/O request is to verify that the I/O request specifies a valid process I/O channel. The process I/O channel is an entry in a system-maintained process table that describes a path of reference from a process to a peripheral device unit. Before a program requests I/O to a device, the program identifies the target device unit by issuing an Assign-I/O-Channel ($ASSIGN) system service call. The $ASSIGN system service performs the following functions:

- Locates an unused entry in the table of process I/O channels
- Creates a pointer to the device unit in the table entry for the channel
- Returns a channel-index number to the program

When the program issues an I/O request, EXE$QIO verifies that the channel number specified is associated with a device and locates the unit control block associated with the specified channel using the field CCB$L_UCB.

Refer to the *VMS Device Support Reference Manual* data structures for an illustration of the channel control block (CCB) and a description of its contents.

4.1.2 Locating a Device Driver in the I/O Database

A unit control block (UCB) that describes a device unit exists for each device in the system. The UCB indicates the current state of the device unit by recording such information as the following:

- Whether the device is active (UCB$V_BSY in UCB$L_STS)
- What I/O request is being processed (UCB$L_IRP)
- Where transfer buffers are located (UCB$L_SVAPTE)

Because drivers run as fork processes and cannot use process address space to store additional context, drivers use the UCB for temporary data storage during I/O processing. (Section 6.1 describes how you can allocate additional UCB space for storing data or device-dependent driver context.)

The UCB also holds the context of a driver fork process when VMS I/O routines suspend the fork process to wait for an asynchronous event such as a device interrupt.

Using information in the UCB, a driver can find other I/O data structures associated with the device, including the channel request block, interrupt dispatch block, and the device data block.

Refer to the *VMS Device Support Reference Manual* data structures for an illustration of the UCB and description of its contents.

4.1.2.1 Channel Request Block

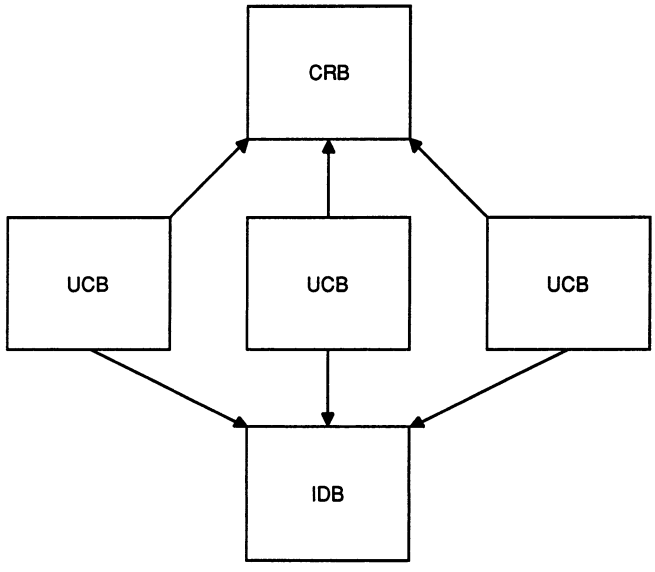
The channel request block (CRB) allows the operating system to manage the controller data channel. Among its contents are the following:

- Code that transfers control to a driver's interrupt service routine (CRB$L_INTD)
- A pointer to the driver's interrupt service routine (CRB$L_INTD+VEC$L_ISR)
- Addresses of a driver's unit and controller initialization routines (CRB$L_INTD+VEC$L_UNITINIT, CRB$L_INTD+VEC$L_INITIAL)
- A pointer to the interrupt dispatch block (IDB), which further describes the controller (CRB$L_INTD+VEC$L_IDB)

Controllers can be either multiunit or dedicated.

All UCBs describing device units attached to a single **multiunit controller** contain a pointer to a single CRB (UCB$L_CRB). For these controllers, a VMS routine uses fields in the CRB (CRBL_WQFL, CRBB_MASK) and IDB (IDB$L_OWNER) to arbitrate pending driver requests for the controller. When the system grants ownership of a multiunit controller data channel to a driver fork process, the fork process can initiate an I/O operation on a device attached to that controller. Figure 4–3 illustrates the data structures required to describe three devices on a multiunit controller.

Figure 4–3 Data Structures for Three Devices on One Controller



The VMS operating system does not use the CRB to synchronize I/O operations for a **dedicated controller**, as the controller manages but a single device. Nevertheless, the CRB still is present and is used by drivers and operating system routines.

Refer to the *VMS Device Support Reference Manual* data structures for an illustration of the CRB and a description of its contents.

4.1.2.2 Interrupt Dispatch Block

The CRB contains a pointer to an interrupt dispatch block (IDB) (CRB$L_INTD+VEC$L_IDB). In turn, the IDB (at IDB$L_UCBLST) points to all UCBs that share the controller (see Figure 4–3).

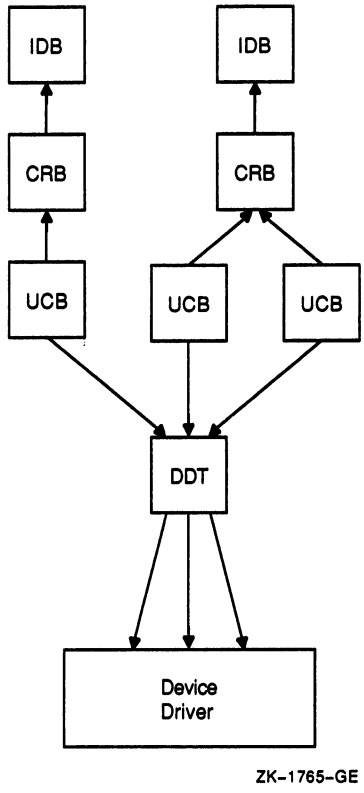
The IDB contains the addresses of these three critical data structures:

- The UCB of the device unit, if any, that currently owns the controller data channel (IDB$L_OWNER)
- The control and status register (IDB$L_CSR); it is the key to access to device registers
- The adapter control block (IDB$L_ADP) that describes the adapter of the I/O bus to which the controller is attached

Refer to the *VMS Device Support Reference Manual* data structures for an illustration of the IDB and a desription of its contents.

Figure 4–4 illustrates the relationship between the data structures that describe a group of equivalent devices on two separate controllers. In this figure, one controller has a single device unit, and the other controller has two device units. Devices on both controllers share the same driver code.

Figure 4–4 I/O Database for Two Controllers



4.1.2.3 Device Data Block

All UCBs describing device units attached to a single controller contain a pointer (UCB$L_DDB) to a single device data block (DDB). The DDB contains two fields that identify the device and its driver:

- The generic device/controller name (DDB$T_NAME)
- The name of the device's driver as obtained from the driver prologue table (DDB$T_DRVNAME)

Refer to the *VMS Device Support Reference Manual* data structures for an illustration of the DDB and description of its contents.

4.1.3 Validating the I/O Function

Using the I/O database, EXE$QIO locates the address of the driver's function decision table by following a chain of pointers that begins in the UCB of the target device:

UCB → DDT → FDT

EXE$QIO then uses data in the function decision table to analyze the I/O function. The procedure confirms that the function specified in the I/O request is a valid function for the device.

4.1.4 Checking Process I/O Request Quotas

EXE$QIO determines whether the I/O request being readied will cause the process to exceed its quota for outstanding direct or buffered I/O requests. If the process's requests remain under quota, the system service allows it to continue I/O preprocessing. Where quota is exceeded, the procedure examines the process's resource wait flag (PCB$V_SSRWAIT in PCB$L_STS).

If the flag is clear, EXE$QIO aborts the I/O request. However, if the flag is set, it places the process in a wait state until previously issued I/O requests complete and the number of requests drops below quota. When this occurs, process execution resumes, at which time EXE$QIO charges process quotas as appropriate for the requested operation.

4.1.5 Validating the I/O Status Block

If the I/O request specifies a quadword I/O status block to receive final I/O status information, EXE$QIO determines whether the process issuing the request has write access to the status block locations specified. If the process has write access, EXE$QIO fills the quadword with zeros. If the process does not have write access, the procedure terminates the request with an error status.

4.1.6 Allocating and Setting Up an I/O Request Packet

If validation of the I/O request succeeds to this point, EXE$QIO allocates a block of nonpaged pool to contain an I/O request packet (IRP).

Before EXE$QIO allocates an IRP, it raises the IPL of the processor to IPL$_ASTDEL to block any other asynchronous activity in the process. The new IPL prevents possible deletion of the process; process deletion would result in the operating system's losing track of the pool allocated for the IRP.

EXE$QIO attempts to allocate an IRP from a **lookaside list** containing preallocated IRPs. If no preallocated packets exist, the procedure calls a VMS routine that allocates an IRP from general nonpaged pool. This allocating routine synchronizes with the rest of the system so that it can allocate the memory needed.

EXE$QIO resumes I/O preprocessing by writing a description of the I/O request into the fields of the IRP, as shown in Table 4–1. Note that this data encompasses the *device-independent* information associated with the request. It is up to the device driver's FDT routines or VMS common FDT routines to fill in the *device-dependent* portions of the IRP, as described in Section 4.1.7 and Chapter 7.

Refer to the *VMS Device Support Reference Manual* data structures for an illustration of the IRP and description of its contents.

Table 4–1 IRP Data Fields

Data	Fields
Size in bytes of the IRP	IRP$W_SIZE
Identification of the block as an IRP	IRP$B_TYPE
Access mode of the process at the time of the request	IRP$B_RMOD
Process ID of the requesting process	IRP$L_PID
Address of an AST routine (if specified in the request) and its parameter[1]	IRPL_AST, IRPL_ASTPRM
For file-structured devices, address of a window control block (WCB) that describes the physical location of part of the file	IRP$L_WIND
Address of the target device's UCB	IRP$L_UCB
I/O function code[2]	IRP$W_FUNC
Number of event flag to set when processing of the I/O request is complete	IRP$B_EFN
Base software priority of the requesting process	IRP$B_PRI
Address of an I/O status block (if specified in the request)	IRP$L_IOSB
Process I/O channel index number	IRP$W_CHAN
A flag indicating whether the I/O function is for buffered or direct I/O	IRP$V_BUFIO in IRP$W_STS
A flag indicating whether the I/O request is an input request	IRP$V_FUNC in IRP$W_STS
A flag indicating whether the I/O function is a physical-I/O function	IRP$V_PHYSIO in IRP$W_STS
Address of a diagnostic buffer (if specified in the request)[3] and a flag indicating that the buffer is present	IRP$L_DIAGBUF, IRP$V_ DIAGBUF in IRP$W_STS
Address of process's access rights block	IRP$L_ARB
I/O transaction sequence number	IRP$L_SEQNUM

[1] If the request specifies an AST, EXE$QIO also verifies that the request would not cause the process to exceed its AST quota. If it would, EXE$QIO aborts the request.

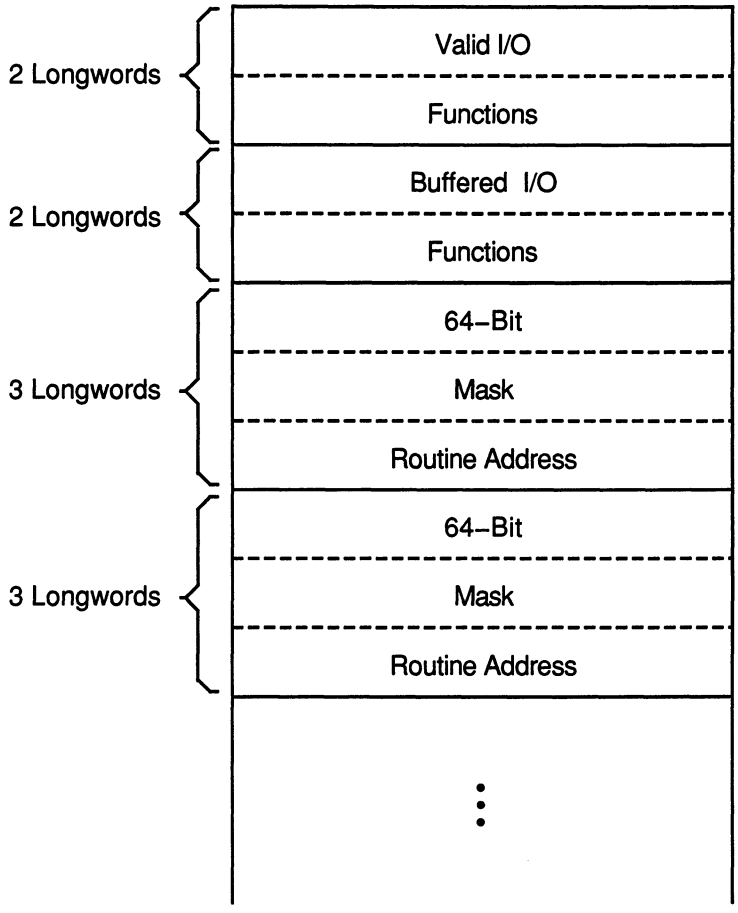
[2] For nonfile devices (DEV$V_FOD clear in UCB$L_DEVCHAR), EXE$QIO reduces read- and write-virtual-block functions to their equivalent read- and write-logical-block functions before storing a code.

[3] The size of the diagnostic buffer is specified in the driver dispatch table of the driver servicing the device unit to which the request is made. See Section 6.2 for more information.

4.1.7 FDT Processing

The driver's function decision table controls the device-dependent preprocessing of an I/O request. Figure 4–5 illustrates the layout of a function decision table.

Figure 4–5 Layout of a Function Decision Table



The I/O function code specified in an I/O request is a 16-bit value consisting of two fields:

- A 6-bit I/O function code (bits 0 through 5) that permits you to define 64 unique I/O function codes for every device type. Table 6–1 lists the function codes defined by VMS. Section 6.3.2 describes how you can define device-specific function codes.
- A 10-bit I/O function modifier (bits 6 through 15). In subsequent processing of the I/O request, the driver's start-I/O routine uses both I/O function code and I/O function modifier, as stored in IRP$W_FUNC, to create a device-specific function code to use in device activation.

The first two entries of a function decision table are two longwords (64 bits) each. The first quadword entry is the **legal function bit mask** of all I/O function codes that are valid for the device. The second quadword entry is the **buffered function bit mask** of those valid I/O functions that are also buffered-I/O functions.

EXE$QIO uses the value of the low-order six bits of the I/O function code to determine which bit to check in each of these bit masks. For example, if the function code has a value of 22, the procedure checks the twenty-third bit (bit 22) of each bit mask. Thus, EXE$QIO determines whether the I/O function code is valid for the device and is able to charge against the appropriate quota of the requesting process for a direct- or buffered-I/O operation.[1]

Subsequent entries in the function decision table are three longwords long, and it is these entries that EXE$QIO uses to dispatch to the appropriate I/O preprocessing routine (FDT routine) for the requested function. Again, the first quadword is a 64-bit bit mask, and is checked by EXE$QIO in exactly the same way as the legal function bit mask and the buffered function bit mask. These **action routine bit masks**, however, contain the address of an FDT routine in the subsequent longword, and it is to this FDT routine that EXE$QIO transfers control when it discovers the bit corresponding to the I/O function set in the quadword.

Some FDT routines are present in the operating system because they provide common services for many devices. Section 7.5 describes these routines. Other routines are included in the device driver because they perform device-dependent services.

EXE$QIO uses the action routine bit mask entries in the function decision table to call FDT routines in the driver or system, according to the following strategy:

1 If the bit corresponding to the function code is set in the action routine bit mask, EXE$QIO calls the FDT routine whose address appears in the following longword.

- If this I/O function requires additional preprocessing after this particular FDT routine completes its activity, the FDT routine returns control to EXE$QIO with an RSB instruction. When EXE$QIO regains control, it advances to the next action routine bit mask and repeats step 1.
- If this FDT routine completes all necessary preprocessing for this particular I/O function, then it transfers control to a VMS routine that queues the IRP or completes the request.

2 If the bit corresponding to the function code is not set, EXE$QIO advances to the next action routine bit mask in the table and repeats step 1.

Note: A single function decision table can specify that EXE$QIO call more than one FDT routine to perform the many and varied steps in the preprocessing of a single I/O function. However, it is the responsibility of the FDT routine that ultimately completes the preprocessing to end the scan (by EXE$QIO) of the function decision table. An FDT routine accomplishes this by transferring control to either a VMS routine that queues the I/O request for the driver's start-I/O routine or one that completes or aborts

[1] For physical- and logical-I/O operations, EXE$QIO also verifies that the process making the I/O request has suitable privileges.

the request (see Figure 4–6). In other words, for each valid I/O function code for a device, an FDT entry must contain the address of a routine that ends I/O preprocessing.

The execution flow of FDT routines is illustrated in Figure 4–6. FDT routines execute in the context of the process that requested the I/O operation. Thus, FDT routines can access process virtual address space. Once all FDT preprocessing is complete, however, the rest of the processing for the I/O request continues in the limited context of a driver fork process or an interrupt service routine.

4.2 Handling Device Activity

When I/O preprocessing is complete, the last-called FDT routine generally jumps (with a JMP instruction) to a routine called EXE$QIODRVPKT.[2] EXE$QIODRVPKT, in turn, transfers control (using a JSB instruction) to EXE$INSIOQ, the VMS routine that queues IRPs and arbitrates device activity. (See Figure 4–2 for a representation of the flow of I/O request processing at this juncture.)

4.2.1 Creating a Driver Fork Process to Start I/O

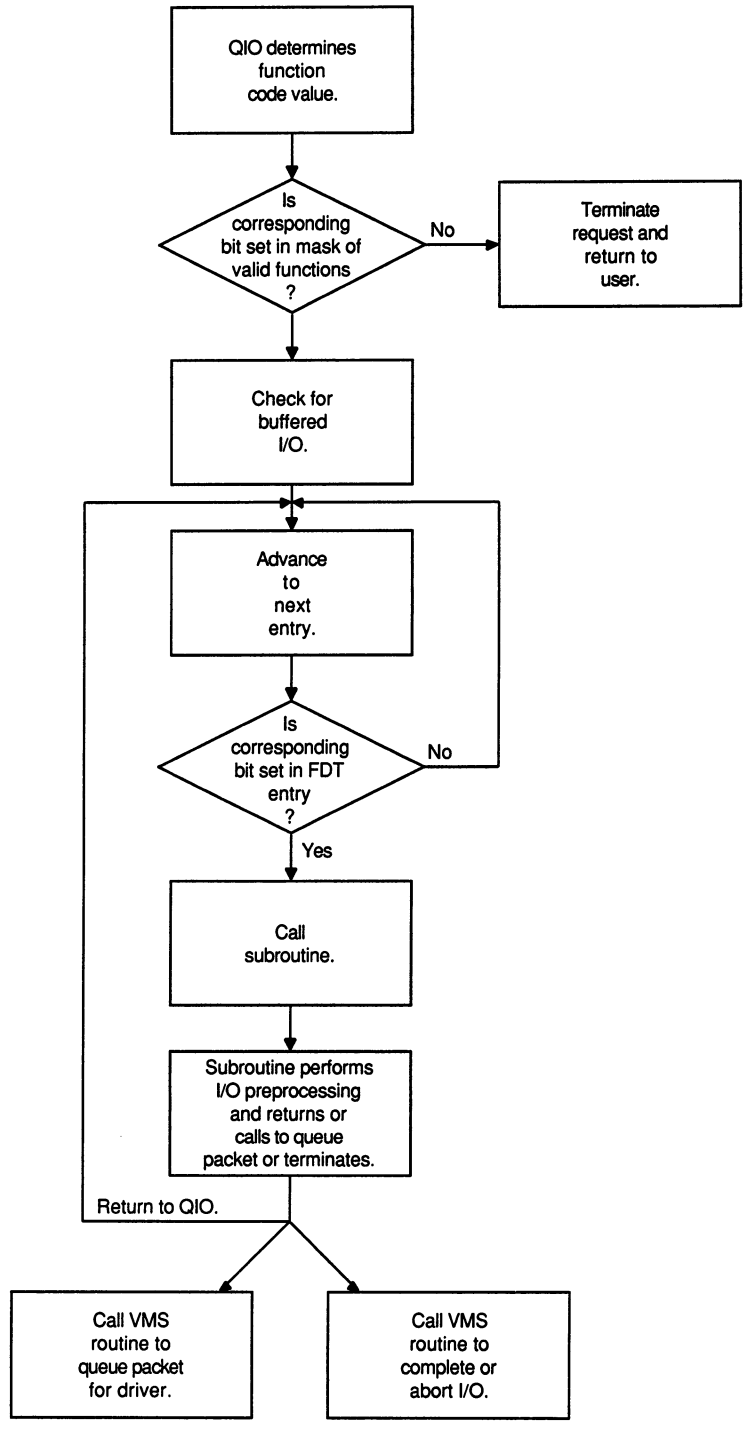
EXE$INSIOQ creates only one driver fork process at a time for each device unit on the system. As a result, only one IRP for each device unit is serviced at one time. EXE$INSIOQ determines whether a driver fork process exists for the target device, as follows:

- If the device is idle, no driver fork process exists for the device; in this case, EXE$INSIOQ immediately calls IOC$INITIATE to create and transfer control to a driver fork process to execute the driver's start-I/O routine.
- If the device is busy, a driver fork process already exists for the device, servicing some other I/O request. In this case, EXE$INSIOQ calls EXE$INSERTIRP to insert the IRP into a queue of IRPs waiting for the device unit. The routine queues the IRP according to the base priority of the caller. Within each priority, IRPs are in first-in/first-out order. The completion of the current I/O request triggers the servicing of the I/O request that is first in the queue, according to the procedure described in Section 10.1.2.3.

In the latter case, by the time the driver's start-I/O routine gains control to dequeue the IRP, the originating user's process context is no longer available. Because the context of the process initiating the I/O request is not guaranteed to a driver's start-I/O routine, the driver must execute in the reduced context available to a fork process.

[2] The rules for exiting from FDT preprocessing, including descriptions of EXE$QIODRVPKT and other FDT exit routines, appear in Section 7.2.

Figure 4–6 FDT Routines and I/O Preprocessing



IOC$INITIATE always initiates the driver's start-I/O routine with a context that is appropriate for a fork process. VMS establishes this context by performing the following steps:

1 Raising IPL to driver fork IPL (and obtaining the associated fork lock in a VMS multiprocessing environment)

2 Loading the address of the IRP into R3

3 Loading the address of the device's UCB into R5

4 Transferring control (with a JMP instruction) to the entry point of the device driver's start-I/O routine

The newly activated driver fork process executes under the constraints listed in Section 3.3.3.2. It executes until one of the following events occurs:

- Device-dependent processing of the I/O request is complete.
- A shared resource needed by the driver is unavailable, as described in Section 3.4.
- Device activity requires the fork process to wait for a device interrupt.

4.2.2 Activating a Device and Waiting for an Interrupt

Depending on the device type supported by the driver, the start-I/O routine performs some or all of the following steps:

1 Analyzes the I/O function and branches to driver code that prepares the UCB and the device for that I/O operation

2 Copies the contents of fields in the IRP into the UCB

3 Tests fields in the UCB to determine whether the device and/or volume mounted on the device are valid

4 If the device is attached to a multiunit controller, obtains the controller data channel

5 If the I/O operation is a DMA transfer, obtains I/O adapter resources such as map registers and a UNIBUS adapter buffered data path

6 Raises IPL to device IPL, obtaining the associated device lock in a VMS multiprocessing environment, to synchronize its access to device registers

7 Loads all necessary device registers except for the device's control and status register (CSR)

8 Raises IPL to IPL$_POWER and confirms that a power failure that would invalidate the device operation has not occurred on the local processor

9 Loads the device's CSR to activate the device

10 Invokes a VMS routine (using either the WFIKPCH or WFIRLCH macro) to suspend the driver fork process until a device interrupt or timeout occurs

This routine (IOC$WFIKPCH or IOC$WFIRLCH) expects to find, among the items it inherits on the stack, the driver's fork IPL, as placed there by the start-I/O routine in step 7. As it suspends the driver, IOC$WFIKPCH or IOC$WFIRLCH saves the driver's context in the UCB's fork block. This context consists of the following information:

- The contents of R3 and R4 (UCBL_FR3, UCBL_FR4)
- The implicit contents of R5 as the address of the UCB
- A driver return address (UCB$L_FPC)
- The relative offset to a device timeout handler (calculated from UCB$L_FPC and the value specified in the invocation of the WFIKPCH or WFIRLCH macro)
- The time at which the device will time out (UCB$L_DUETIM)

By convention, R4 often contains the address of the CSR; it permits the driver to examine device registers. When the driver fork process regains control after interrupt processing, R5 contains the UCB address; it is the key to the rest of the I/O database that is relevant to the current I/O operation.

Having removed the driver's start-I/O routine's return address from the stack and stored it in UCBL_FPC, IOCWFIKPCH (or IOC$WFIRLCH) issues a DEVICEUNLOCK macro that restores IPL to fork IPL from the stack. It then exits with an RSB instruction. Thus, IOC$WFIKPCH (or IOC$WFIRLCH) effectively passes control to the caller of its caller. In this case, the caller of the driver start-I/O routine is EXE$INSIOQ. The flow back from EXE$INSIOQ to a user process that asynchronously requested the I/O operation is shown in Figure 4–2.

You can find additional information on the context of a start-I/O routine in Chapter 8.

4.2.3 Handling a Device Interrupt

When the device requests an interrupt, the interrupt dispatcher transfers control to the driver interrupt service routine. The driver's interrupt service routine runs at a high IPL so that the routine can service interrupts quickly. A driver interrupt service routine usually performs the following processing:

1 Retrieves the address of the UCB that owns the controller from IDB$L_OWNER

2 Issues the DEVICELOCK macro to obtain the device lock associated with operations at device IPL in a VMS multiprocessing environment

3 For multiunit device controllers, determines which device unit generated the interrupt

4 Examines the UCB for the device to confirm that the driver fork process expects the interrupt

5 Saves device registers

6 Reactivates the suspended driver fork process

If necessary, the reactivated driver fork process executes at the high IPL of the interrupt service routine for a few instructions. Very soon, however, the driver lowers its execution priority so that it does not block subsequent interrupts for other devices in the system.

4.2.4 Switching from Interrupt to Fork Process Context

To lower its priority, the driver calls a VMS fork process queuing routine (by means of the IOFORK macro) that performs the following actions:

1 Disables the timeout that was specified in the wait-for-interrupt routine

2 Saves R3 and R4 (UCBL_FR3, UCBL_FR4)

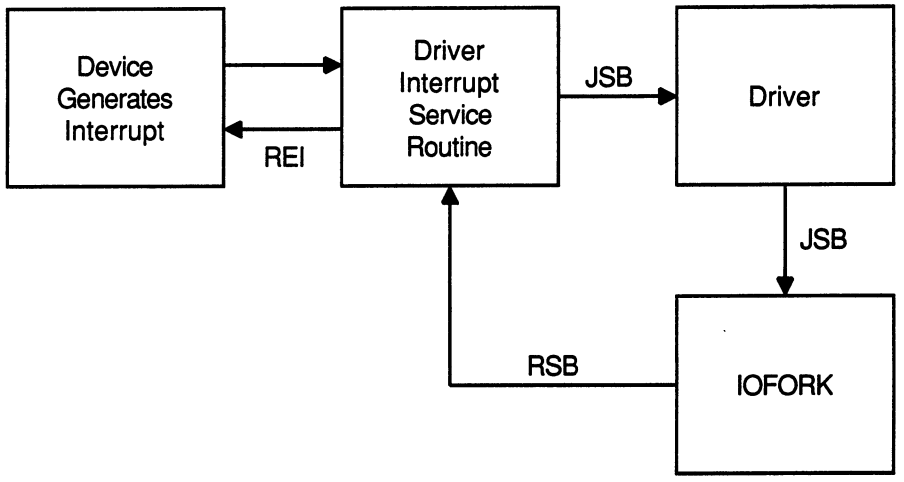
3 Saves the address of the instruction following the IOFORK request in the UCB fork block (UCB$L_FPC)

4 Places the address of the UCB fork block from R5 in a processor-specific fork queue for the driver's fork level

5 Returns to the driver's interrupt service routine

The interrupt service routine then cleans up the stack, issues the DEVICEUNLOCK macro to release the device lock, restores registers, and dismisses the interrupt. Figure 4–7 illustrates the flow of control in a driver that creates a fork process after a device interrupt.

Figure 4–7 Creating a Fork Process After an Interrupt



4.2.5 Activating a Fork Process from a Fork Queue

When no higher priority interrupts are pending, the local processor transfers control to the fork dispatcher. When the processor grants an interrupt at a fork IPL, the fork dispatcher processes the local fork queue that corresponds to the IPL of the interrupt. To do so, the dispatcher performs these actions:

1 Removes a fork block from the fork queue

2 Restores fork context

3 Obtains the fork lock specified in the fork block

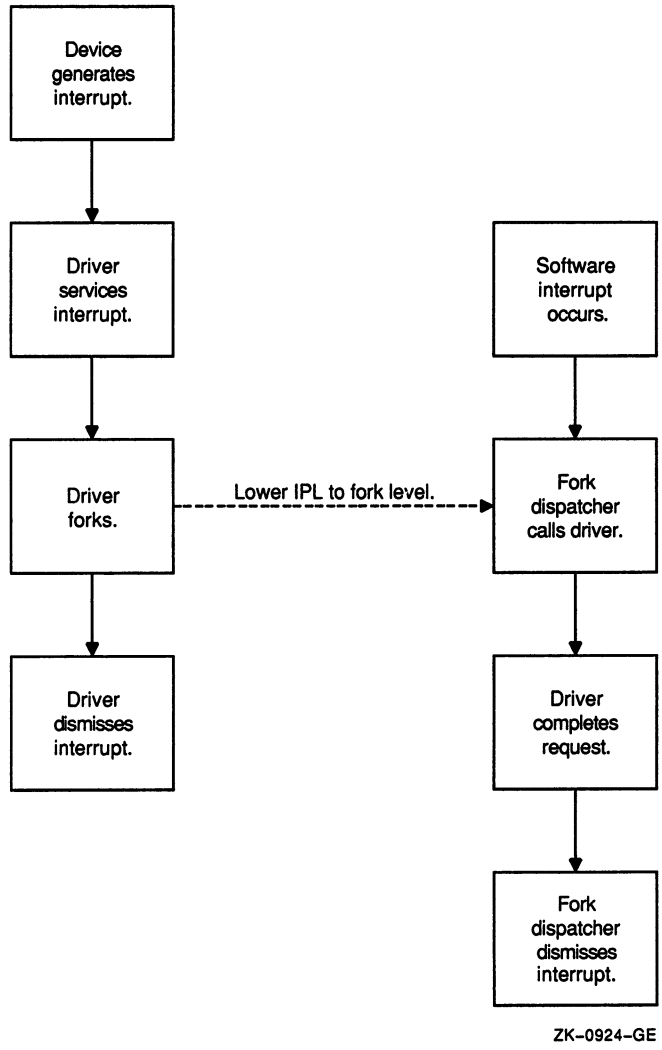
4 Transfers control back to the fork process

Thus, the driver code calls VMS code that coordinates suspension and restoration of a driver fork process. This convention allows VMS to service hardware device interrupts in a timely manner and reactivate driver fork processes as soon as no device requires attention.

When a given fork process completes execution, the fork dispatcher releases the fork lock and removes the next entry, if any, from the local fork queue. This fork dispatcher repeats the sequence described previously until the fork queue is empty. After servicing the last entry in the queue, the fork dispatcher releases the fork lock, restores R0 through R5 from the stack, and dismisses the interrupt with an REI instruction.

Figure 4–8 illustrates the reactivation of a driver fork process.

Figure 4–8 Reactivation of a Driver Fork Process



4.3 Completing an I/O Request

Once reactivated, a driver fork process completes the I/O request as follows:

1 Releases shared driver resources, such as map registers, UNIBUS adapter buffered data path, and controller ownership

2 Returns status to the VMS I/O completion routine

The I/O-completion routine performs the following steps to start postprocessing of the I/O request and to start processing the next I/O request in the device's queue:

1 Writes return status from the driver into the IRP

2 Inserts the finished IRP in the systemwide I/O postprocessing queue and requests an interrupt from the processor at IPL$_IOPOST

3 Creates a new fork process for the next IRP in the device's pending-I/O queue

4 Activates the new driver fork process

4.3.1 I/O Postprocessing

When the local processor's IPL drops below the I/O postprocessing IPL, the processor dispatches to the I/O postprocessing interrupt service routine. This VMS routine completes device-independent processing of the I/O request.

Using the IRP as a source of information, the IPL$_IOPOST dispatcher executes the following sequence for each IRP in the postprocessing queue:

1 Removes the IRP from the queue

2 If the I/O function was a direct I/O function, adjusts the issuing process's direct I/O quota and unlocks the pages involved in the I/O transfer

3 If the I/O function was a buffered I/O function, adjusts the issuing process's buffered I/O quota and, if the I/O was a write function, deallocates the system buffers used in the transfer

4 Posts the local event flag associated with the I/O request

5 Queues a special kernel-mode AST routine to the process that issued the $QIO system service call

The queuing of a special kernel-mode AST routine allows I/O postprocessing to execute in the context of the user process but in a privileged access mode. Process context is needed to return the results of the I/O operation to the process's address space. The special kernel-mode AST routine sets any common event flag associated with the I/O request and writes the following data into the process's address space:

- Data read in a buffered I/O operation
- If specified in the I/O request, the contents of the diagnostic buffer
- If specified in the I/O request, the two longwords of I/O status

If the I/O request specifies an I/O completion AST routine, the special kernel-mode AST routine queues the I/O completion AST for the process. When VMS delivers the I/O completion AST, the system AST delivery routine deallocates the IRP. The first part of an IRP is the AST control block for user requested ASTs.

5 Device Driver Coding Format

This chapter describes the coding format of a basic driver program. It describes basic program flow and code conventions to follow in the initial phase of device driver programming.

A VMS template driver is provided in Appendix A and a VMS SCSI class driver template is provided in Appendix B. The code in the VMS template can serve as a starting point for coding a new UNIBUS or Q22 bus device driver. The code in the SCSI template can serve as the starting point for coding a new SCSI class driver. You can obtain machine-readable copy of these templates from SYS$EXAMPLES:TDRIVER.MAR and SYS$EXAMPLES:SKDRIVER.MAR, respectively.

Drivers do not necessarily need all of the routines indicated by the templates, nor do driver routines and tables need to follow the exact order of the templates. However, the VMS operating system does place a few restrictions on the order and content of driver routines and tables.

Figure 5–1 illustrates the organization of a device driver. The first item in a device driver is the driver prologue table and the second is usually the driver dispatch table. The order of the remaining driver components varies from driver to driver.

The last statement in every driver, except for the .END assembly directive, must be a label marking the end of the driver. The address of this label is stored in the driver prologue table. The driver-loading procedure uses this address to calculate the size of the driver. Chapter 12 describes the driver-loading procedure.

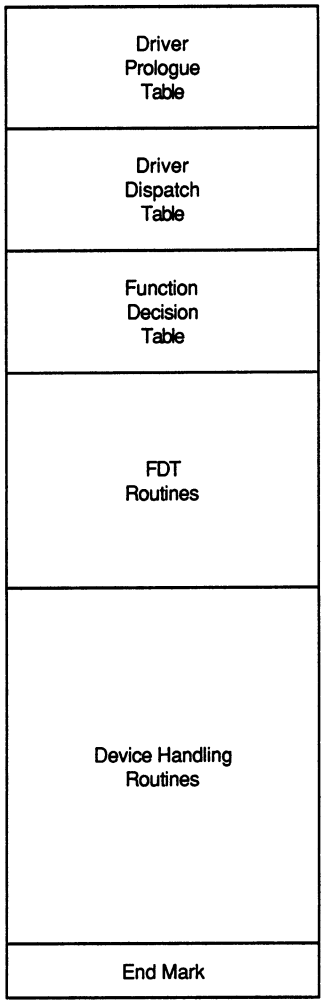
Some drivers contain no device-dependent, FDT routines. Other drivers need only minimal initialization procedures. However, every driver normally contains static driver tables and a start-I/O routine or an interrupt service routine.

5.1 Coding Conventions

The driver-loading procedure loads a device driver into a block of nonpaged system memory whose location is chosen by the operating system memory allocation routines. Therefore, the driver must consist of position-independent code only.

In addition, the system might call a device driver repeatedly to process I/O requests and interrupts. The driver often does not complete one I/O operation before the system transfers control to the driver to begin another on a different unit. For this reason, the code must be reentrant.

Figure 5–1 Driver Organization



The rules of position-independent and reentrant code are as follows:

- Instructions can branch only to relative addresses within the driver and to global addresses listed in the VMS symbol table (SYS$SYSTEM:SYS.STB).
- Static tables can list only global addresses and relative addresses within the driver.
- The driver cannot store temporary data in local driver tables for dynamic driver context. All dynamic temporary storage must be contained within the unit control block corresponding to an I/O request or the current I/O request block.

- The driver must refer to the I/O database by loading the address of a data structure into a general register and using displacement addressing to the fields of the data structure.

Device drivers must also restrict their use of general registers and the stack:

- FDT routines can use R0 through R2 and R9 through R11 as available registers. The routines can use other registers by saving the registers before use and restoring them before exiting from the FDT routine.
- All other driver routines can use R0 through R5 as available registers. The routines can use other registers, if necessary, by saving and restoring them; but using other registers in this way is discouraged.
- All driver routines can use the stack for temporary storage only if the routines restore the stack to its previous state before calling any VMS routines, forking, or executing RSB instructions.

Because certain VAX processors and VMS cooperate to support the emulation of specific sets of VAX instructions, a device driver writer should exercise some caution. Because the software emulation for floating-point instructions may at some time be placed in pageable code, drivers should *never* use floating-point instructions. VMS only guarantees the emulation for character string instructions to be nonpaged.

Finally, the use of VAX vector instructions is prohibited above IPL$_ASTDEL.

5.2 Restrictions on the Use of Device-Register I/O Space

The programmer of a device driver must observe the following restrictions on the use of device registers:

- Drivers should always store the address of a device control register in a general register and then gain access to the device register indirectly through the general register. The following example defines symbolic word offsets for each device register and gains access to them using displacement-mode addressing from R4.

```
;
; Device register offsets
;
LP_CSR = 0                                  ;CSR offset
LP_DBR = 2                                  ;Buffer address offset

    .
    .
    .
    MOVL    UCB$L_CRB(R5),R4                ;Get address of CRB
    MOVL    @CRB$L_INTD+VEC$L_IDB(R4),R4    ;Get the address of
                                            ; the device's CSR
      .
      .
      .
    TSTW    LP_CSR(R4)                      ;Is printer on line?
```

- Floating-point, field, queue, quadword, octaword, and vector operands are not allowed in I/O address space, nor can an instruction obtain the position, size, length, or base of an operand from I/O space. For example, a driver cannot use a bit field instruction to test a bit in a device register.
- Drivers cannot use string-handling instructions when referring to I/O space.
- Drivers can use only those instructions that modify or write to a maximum of one destination. The destination must be the last operand.
- Registers of devices connected to the backplane interconnect (for example, UNIBUS adapter device registers and MASSBUS device registers) are longwords. Registers of devices connected to the UNIBUS or Q22 bus are words. Instructions that refer to UNIBUS adapter registers must use longword context. All driver instructions that affect UNIBUS or Q22 bus device registers must use word context (for example, BISW, MOVW, and ADDW3) unless the register is byte addressable.
- An instruction that refers to I/O space must not generate an exception or be interruptable. If the instruction is allowed to restart, it will reread the device register, which can cause undesirable device side effects or data loss.
- On any given VAX processor, a device driver cannot anticipate the completion of an instruction that writes to I/O space before subsequent instructions execute. The processor can continue to execute without waiting for the data to reach its intended destination.

  Among the consequences of this behavior are the following:

  — If a driver initiates device actions that result in an interrupt from the device, the amount of time before that interrupt actually occurs is unpredictable.

  — If a driver disables interrupts from a device, the time before that device can no longer generate an interrupt is unpredictable.

  — An I/O bus error will not be reported synchronously with the instruction causing the error.

  As a result, a driver's interrupt service routine always should be prepared to service unexpected or spurious interrupts. See Section 9.3 for additional discussion of the servicing of unexpected interrupts.

- To access I/O space, use only the following instructions. These instructions cannot be interrupted unless they use autoincrement-deferred addressing mode or any of the displacement-deferred modes when specifying an operand.

ADAWI	ADD(B,W,L)2	ADD(B,W,L)3
ADWC	BIC(B,W,L)2	BIC(B,W,L)3
BICPSW	BIS(B,W,L)2	BIS(B,W,L)3
BISPSL	BISPSW	BIT(B,W,L)
CASE(B,W,L)	CHM(K,E,S,U)	CLR(B,W,L)
CMP(B,W,L)	CVT(BW,BL,WB, WL,LB,LW)	DEC(B,W,L)
INC(B,W,L)	MCOM(B,W,L)	MFPR
MNEG(B,W,L)	MOV(B,W,L)	MOVA(B,W,L)
MOVAQ	MOVPSL	MOVZ(BW,BL,WL)
MTPR	PROBE(R,W)	PUSHA(B,W,L)
PUSHAQ	PUSHL	SBWC
SUB(B,W,L)2	SUB(B,W,L)3	TST(B,W,L)
XOR(B,W,L)2	XOR(B,W,L)3	

5.3 Implementing Conditional Code in a Driver

When writing a DMA driver to function for equivalent devices on different I/O bus implementations, you should use the ADPDISP macro in code paths that need to differentiate between the systems.

The ADPDISP macro (defined in SYS$LIBRARY:LIB.MLB) provides a means by which a device driver can be designed to drive a similar device in a variety of VAX configurations. The ADPDISP macro allows the driver to determine at run time the existence of a certain I/O bus or adapter characteristic, and transfer control to code designed to execute given this hardware trait.

A driver can use ADPDISP to transfer control to specific code given any of the following characteristics:

- Adapter type
- Number of adapter address bits (18 or 22)
- Map registers supported
- Autopurging data paths supported
- Buffered data paths supported
- Direct-vector interrupt dispatching supported
- Odd-aligned transfers on buffered data path supported
- Odd-aligned transfers on direct data path supported
- Alternate set of map registers (496 to 8191) available
- Q22 bus device

Use ADPDISP when it is necessary to conditionally execute pieces of code, for instance, the allocation and loading of map registers for devices for which map registers are available or the allocation of a physically contiguous buffer for a DMA transfer on a generic VAXBI device which requires such a buffer). VMS supplies a similar macro, CPUDISP, which causes a run-time transfer of control to a specified destination depending on the CPU type of the executing processor. For those processors not uniquely identified by CPU type, CPUDISP also provides the means to dispatch on a particular CPU subtype.

Because a device driver cannot make assumptions about the I/O architecture of any given VAX system, Digital recommends that most instances of the CPUDISP macro be replaced by an appropriate usage of the ADPDISP macro.

Appendix C and Appendix D contain examples of drivers that use the ADPDISP macro to provide conditional code in a driver. See also the description of the ADPDISP macro in the macro chapter of the *VMS Device Support Reference Manual*.

6 Writing Device-Driver Tables

Every device driver declares three static tables that describe the device and driver:

- **Driver prologue table**—Describes the device type, driver name, and fields in the I/O database to be initialized during driver loading and reloading.
- **Driver dispatch table**—Lists some of the driver's entry points to which VMS transfers control. The channel request block and function decision table list other entry points.
- **Function decision table**—Lists valid functions of the driver and entry points to routines that perform I/O preprocessing for each function.

The VMS operating system provides macros that drivers can invoke to create these tables.

6.1 Driver Prologue Table

The driver prologue table (DPT) is the first part of every device driver. This table, along with parameters to the SYSGEN command that request driver loading, describes the driver to the driver-loading procedure. In turn, the driver-loading procedure computes the size of the driver, loads it into nonpaged system memory, and creates data structures for the new devices in the I/O database. The loading procedure also links the new DPT into a list of all DPTs known to the system. Chapter 12 describes how the driver-loading procedure decides which data structures to build for a given device.

Device drivers can pass data-structure initialization information to the driver-loading procedure through values stored in the DPT. In addition, the driver-loading procedure initializes some fields within the device data structures using information from its own tables.

The contents of the DPT data structure are shown and described in the *VMS Device Support Reference Manual* data structures section. Drivers must treat many of the fields initialized by the driver-loading procedure as read-only fields. These fields are marked with an asterisk in the *VMS Device Support Reference Manual* DPT illustration.

To create a DPT, the driver invokes the DPTAB macro, as described in the macro chapter of the *VMS Device Support Reference Manual*. The DPTAB macro generates a driver prologue table (DPT) in a program section called $$$105_PROLOGUE.

The DPTAB macro requires the following information:

- Address of the end of the driver in its **end** argument.

- Code identifying the device by its adapter type in the **adapter** argument. Accepted adapter types include UBA (for devices attached to either a UNIBUS or Q22 bus), MBA, and GENBI.
- Name of the driver in the **name** argument. (Digital reserves to customers driver names beginning with the letters *J* and *Q*.)
- Size of the unit control block (UCB) in the **ucbsize** argument. (The template in Appendix A and the macro descriptions in *VMS Device Support Reference Manual* demonstrate how you can specify an extended UCB defined by VMS or create an extended UCB within a driver.)

The DPTAB also allows you to specify the following information, if applicable to the device driver:

- Whether the driver needs a permanently allocated system page
- Whether the driver has been written to run in a VMS symmetric multiprocessing system
- Name of a driver unloading routine, if any, to be called subject to a SYSGEN RELOAD command
- Maximum number of units supported by the driver (default is 8)
- Number of UCBs to be created when the driver is loaded by means of the SYSGEN autoconfiguration facility and the address of a unit delivery routine to be called by that facility

A driver follows the DPTAB macro invocation with several instances of the DPT_STORE macro. The DPT_STORE macro provides the driver with a means of communicating its initialization needs to the driver-loading procedure. When invoked, the DPT_STORE macro places information in the DPT that the driver-loading procedure uses to load specified values into specified fields. The DPT_STORE macro accepts two lists of fields:

- Fields to be initialized only when the driver is first loaded
- Fields to be initialized when a driver is first loaded and reinitialized if the driver is reloaded

The DPTAB macro stores the relative addresses of these two lists, called initialization and reinitialization tables, in the DPT.

Drivers use the DPT_STORE macro with the INIT table marker label to begin a list of DPT_STORE invocations that supply initialization data for the following fields:

UCB$B_FLCK	Index of the fork lock under which the driver performs fork processing. The DPTAB macro, in invoking the $SPLCODDEF macro, defines the symbols for these indexes.
UCB$B_DIPL	Device interrupt priority level.

Other commonly initialized fields are

UCB$L_DEVCHAR	Device characteristics
UCB$B_DEVCLASS	Device class
UCB$B_DEVTYPE	Device type
UCB$W_DEVBUFSIZ	Default buffer size
UCB$Q_DEVDEPEND	Device-dependent parameters

Drivers use the DPT_STORE macro with the REINIT table marker label to begin a list of DPT_STORE invocations that supply initialization and reinitialization data for certain fields. Every driver must specify the following field in such an invocation:

DDB$L_DDT	Driver dispatch table

Other commonly initialized fields are

CRB$L_INTD+VEC$L_ISR	Interrupt service routine.
CRB$L_INTD2+VEC$L_ISR	Interrupt service routine for second interrupt vector.
CRB$L_INTD+VEC$L_INITIAL	Controller initialization routine.
CRB$L_INTD+VEC$L_UNITINIT	Unit initialization routine (for UNIBUS, Q22 bus, and generic VAXBI device drivers). Note that MASSBUS drivers must specify the address of the unit initialization routine in an invocation of the DDTAB macro.

For an example of the use of the DPT and DPT_STORE macros, see the description of the DPTAB macro in *VMS Device Support Reference Manual.*

6.2 Driver Dispatch Table

The driver dispatch table (DDT) lists some of the entry points for driver routines to be called by VMS for I/O processing. Every driver must create a DDT.

The routines listed in the DDT can reside in the driver module or in a VMS module. The routines chapter in the *VMS Device Support Reference Manual* describes the VMS device-independent routines that can be specified.

Device-dependent routines are normally located in the driver module. The DDT contains relative addresses for routines located in the driver module and absolute addresses for routines located in the operating system. At loading time, the driver-loading procedure changes the relative addresses of driver routines to absolute addresses.

The driver creates a DDT by invoking the macro DDTAB. The DDTAB macro labels the DDT devnam$DDT, according to the value you supply in its **devnam** argument. The driver-loading procedure writes the address of the DDT table, as specified in a DPT_STORE macro, into the DDB. Refer to the *VMS Device Support Reference Manual* for an illustration of the DDT and a description of its contents.

The DDTAB macro also generates the program section ($$$115_DRIVER) in which the DDT itself and all driver code reside.

The DDTAB macro has a single required argument, **functb**, for which the driver must specify the address of its function decision table. Several optional arguments allow the driver to specify the names of the following routines, if applicable:

- Start-I/O routine
- Unsolicited interrupt service routine (for MASSBUS device drivers)
- Cancel-I/O routine
- Register dumping routine
- Unit initialization routine
- Alternate start-I/O routine
- Cloned UCB routine

In addition, you specify the length of any diagnostic buffer or error message buffer using the DDTAB macro.

See the description of the DDTAB macro in the *VMS Device Support Reference Manual* for additional information.

6.3 Function Decision Table

The function decision table (FDT) lists codes for I/O functions that are valid for the device; indicates whether the functions are buffered-I/O functions; and specifies routines to perform preprocessing for particular functions. Every device driver must create an FDT containing three or more entries:

- The list of valid I/O function codes
- The list of buffered I/O function codes
- One or more entries each of which specifies all or a subset of I/O function codes and the address of a routine that performs I/O preprocessing for those function codes

If no buffered I/O functions are defined for the device, the second entry contains an empty list.

Taken together, the third through last entries in the FDT specify one or more FDT routines for each valid I/O function code for the device. The FDT routines must terminate the I/O preprocessing for each type of function by transferring control out of the $QIO system service and into a routine that queues the I/O request to a driver, inserts the I/O request in the postprocessing queue, or aborts the I/O request.

Refer to Chapter 7 for information on writing FDT routines.

Table 6–1 lists the physical, logical, and virtual I/O function codes defined by VMS. A complete list of function codes and values is contained in the macro $IODEF in SYS$LIBRARY:STARLET.MLB.

Table 6–1 I/O Function Codes

Function	Description	Equivalent Symbols
Physical I/O		
IO$_NOP[1]	No operation	–
IO$_UNLOAD	Unload drive (required by all disk drivers)	IO$_LOADMCODE
IO$_SEEK	Seek cylinder	IO$_SPACEFILE[1] (space files), IO$_STARTMPROC[1] (start microprocessor)
IO$_RECAL[1]	Recalibrate drive	IO$_DUPLEX[1] (enter duplex mode), IO$_STOP[1] (stop)
IO$_DRVCLR[1]	Drive clear	IO$_INITIALIZE (initialize), IO$_MIMIC[1] (enter mimic mode)
IO$_RELEASE[1]	Release port	IO$_SETCLOCKP[1] (set clock—physical)
IO$_OFFSET[1]	Offset read heads	IO$_ERASETAPE[1] (erase tape), IO$_STARTDATAP[1] (start data transfer—physical)
IO$_RETCENTER[1]	Return to center line	IO$_QSTOP[1] (queue stop request)
IO$_PACKACK	Pack acknowledgment (required by all disk drivers)	–
IO$_SEARCH	Search for sector	IO$_SPACERECORD[1] (space records), IO$_READRCT[1] (read replacement and caching table)
IO$_WRITECHECK	Write check data	–
IO$_WRITEPBLK	Write physical block	–
IO$_READPBLK	Read physical block	–
IO$_WRITEHEAD[1]	Write header and data	IO$_RDSTATS[1] (read statistics), IO$_CRESHAD[1] (create a shadow set)
IO$_READHEAD[1]	Read header and data	IO$_ADDSHAD[1] (add member to shadow set)
IO$_WRITETRACKD[1]	Write track data	IO$_COPYSHAD[1] (perform shadow set copy operations)
IO$_READTRACKD[1]	Read track data	IO$_REMSHAD[1] (remove member from shadow set)
IO$_AVAILABLE	Set device available (required by all disk drivers)	–
IO$_SETPRFPATH[1]	Set preferred path	–
IO$_DISPLAY[1]	Display MSCP/TMSCP volume label	–
IO$_DSE	Data security erase (and rewind)	–
IO$_REREADN[1]	Reread next	–
IO$_REREADP[1]	Reread previous	–
IO$_WRITERET[1]	Write retry	IO$_WRITECHECKH[1] (write check header and data)

[1] Unsupported; subject to change without notice

(continued on next page)

Table 6–1 (Cont.) I/O Function Codes

Function	Description	Equivalent Symbols
Physical I/O		
IO$_READPRESET[1]	Read in preset	IO$_STARTSPNDL[1] (start spindle)
IO$_SETCHAR	Set device characteristics	–
IO$_SENSECHAR	Sense device characteristics	–
IO$_WRITEMARK[1]	Write tape mark	IO$_COPYMEM[1] (copy memory)
IO$_WRTTMKR[1]	Write tape mark retry	IO$_DIAGNOSE[1] (diagnose), IO$_SHADMV[1] (perform mount verification on shadow set)
IO$_FORMAT	Format	IO$_CLEAN[1] (clean tape)
Logical I/O		
IO$_WRITELBLK	Write logical block	–
IO$_READLBLK	Read logical block	–
IO$_REWINDOFF	Rewind and set offline	IO$_READRCTL[1] (read RCT sector 0)
IO$_SETMODE	Set mode	–
IO$_REWIND	Rewind tape	–
IO$_SKIPFILE	Skip files	–
IO$_SKIPRECORD	Skip records	–
IO$_SENSEMODE	Sense mode	–
IO$_WRITEOF	Write end of file	–
IO$_TTY_PORT[1]	Terminal port FDT routine	IO$_FREECAP[1] (return free capacity)
IO$_FLUSH[1]	Flush controller cache	–
IO$_READLCHUNK[1]	Read large logical block	–
IO$_WRITELCHUNK[1]	Write large logical block	–
Virtual I/O		
IO$_WRITEVBLK	Write virtual block	–
IO$_READVBLK	Read virtual block	–
IO$_ACCESS	Access file	–
IO$_CREATE	Create file	–
IO$_DEACCESS	Deaccess file	–
IO$_DELETE	Delete file	–
IO$_MODIFY	Modify file	–
IO$_NETCONTROL[1]	X25 network control function	–
IO$_READPROMPT	Read terminal with prompt	IO$_SETCLOCK (set clock)
IO$_ACPCONTROL	Miscellaneous ACP control	IO$_STARTDATA (start data)

[1] Unsupported; subject to change without notice

(continued on next page)

Table 6–1 (Cont.) I/O Function Codes

Function	Description	Equivalent Symbols
Virtual I/O		
IO$_MOUNT[1]	Mount volume	–
IO$_TTYREADALL[1]	Terminal read passall	–
IO$_TTYREADPALL[1]	Terminal read with prompt passall	–
IO$_CONINTREAD	Connect to interrupt read-only	–
IO$_CONINTWRITE	Connect to interrupt with write	–

[1] Unsupported; subject to change without notice

The device driver creates an FDT by invoking the FUNCTAB macro. Each invocation of the FUNCTAB macro creates a 2- or 3-longword entry in the FDT. The first two invocations create 2-longword entries because they specify only function codes; they do not specify an accompanying action routine.

All subsequent invocations of the FUNCTAB macro must specify both function codes and the address of a routine that is to perform preprocessing for those functions. These invocations create 3-longword entries.

The $QIO system service processes entries in the order in which they appear in the FDT. When a function code is present in more than one 3-longword entry, the system service sequentially calls every routine specified for the function code until a routine stops the scan by aborting, completing, or queuing an I/O request.

See the description of the FUNCTAB macro, and the example of its use, in the *VMS Device Support Reference Manual* for additional information on creating an FDT.

6.3.1 Defining Buffered-I/O Functions

The second entry in an FDT is a **buffered function bit mask** that indicates which legal functions the driver handles as buffered-I/O operations. In selecting the functions that are to be buffered, you should take the following information into consideration:

- Direct I/O is intended only for devices whose I/O operations always complete quickly. For example, although terminal I/O appears fast, users can prevent the I/O operation from completing by using Ctrl/S to halt the operation indefinitely; therefore, terminal I/O operations are buffered I/O.
- Use of direct I/O requires that the process pages containing the buffer be locked in memory. Locking pages in memory increases the overhead of swapping the process that contains the pages.

- Use of buffered I/O requires that the data be moved from the system buffer to the user buffer. Moving data requires additional time.
- Routines that manipulate data before delivering it to the user (for example, an interrupt service routine for a terminal) cannot gain access to the data if direct I/O is used. Therefore, transfers that require data manipulation must be buffered I/O.
- VMS handles the quotas differently for direct I/O and buffered I/O, as described in the *Guide to Maintaining a VMS System*.
- Generally, direct-memory-access (DMA) devices use direct I/O, while programmed I/O devices use buffered I/O.

6.3.2 Defining Device-Specific Function Codes

You can also define device-specific function codes by equating the name of a device-specific function with the name of an existing function that is irrelevant to the device. The selected codes should, however, have a type (logical, physical, or virtual) that is appropriate for the function they represent. Also, user programs that issue $QIO requests specifying a device-specific code must similarly redefine the existing function. For example, the assembly code that follows defines three device-specific physical I/O function codes.

```
IO$_STARTCLOCK=IO$_ERASETAPE      ; Start interval clock
IO$_STOPCLOCK=IO$_OFFSET          ; Stop interval clock
IO$_STARTDATA=IO$_SPACEFILE       ; Start data acquisition
```

7 Writing FDT Routines

The $QIO system service uses the driver's function decision table (FDT) to determine which FDT routines to call to preprocess an I/O request. These FDT routines validate process-specified arguments to the $QIO request. VMS supplies many device-independent FDT routines. Device drivers contain device-dependent FDT routines.

A driver should call the VMS device-independent FDT routines, described in Section 7.5, whenever possible. This practice encourages the use of well debugged routines and minimizes driver size.

7.1 Context of FDT Routine Execution

The $QIO system service executes in the context of the process that issues the I/O request, but in kernel mode and at IPL$_ASTDEL. The process is executing in kernel mode because the dispatching of the $QIO system service executes a CHMK instruction. Process context allows the $QIO system service and driver FDT routines to access process address space. Because the $QIO system service expects FDT routines to preserve this context, an FDT routine observes the following conventions:

- It cannot call VMS system services or VMS RMS services.
- It does not lower IPL below IPL$_ASTDEL. If a routine raises IPL, it must obtain any appropriate spin lock, and it must lower IPL to IPL$_ASTDEL before exiting, releasing any acquired spin lock.
- It does not alter the stack without restoring its original state before exiting.
- If it issues a subroutine call, it must preserve the contents of R3 through R8 across the call. It can, however, use R0 through R2 and R9 through R11 without saving their previous contents. If an FDT routine needs to use R3 through R8, it can use the PUSHR and POPR instructions to save registers on the stack and later restore them.
- It exits either by an RSB instruction to return control to the system service, or it issues a JMP instruction to one of the VMS routines described in Section 7.2.1.

Before calling an FDT routine, the $QIO system service sets up the contents of certain registers, as described in Table 7–1.

Table 7–1 Registers Loaded by the $QIO System Service

Register	Content
R0	Address of FDT routine being called
R3	Address of IRP for current I/O request
R4	Address of process control block (PCB) of current process
R5	Address of UCB of device assigned to user-specified process-I/O channel
R6	Address of CCB that describes user-specified process-I/O channel
R7	Bit number of user-specified I/O function code
R8	Address of current entry in FDT
AP	Address of first function-dependent argument (**p1**) specified in I/O request

While FDT routines can perform extensive preprocessing, such as determining whether user buffers are accessible and reformatting data into buffers in the system address space, they should not access device registers because the device might be active. Furthermore, FDT routines should exercise restraint when modifying the UCB. Routines usually access the UCB while holding the associated fork lock at driver fork IPL to synchronize modifications, and FDT routines do not execute with such synchronization. Drivers containing FDT routines that access device registers or carelessly modify the UCB risk unpredictable operation or a system failure.

7.2 FDT Routines and Their Exit Paths

To transfer control to an FDT routine, the $QIO system service loads the address of the FDT routine into a register and executes a JSB instruction, as follows:

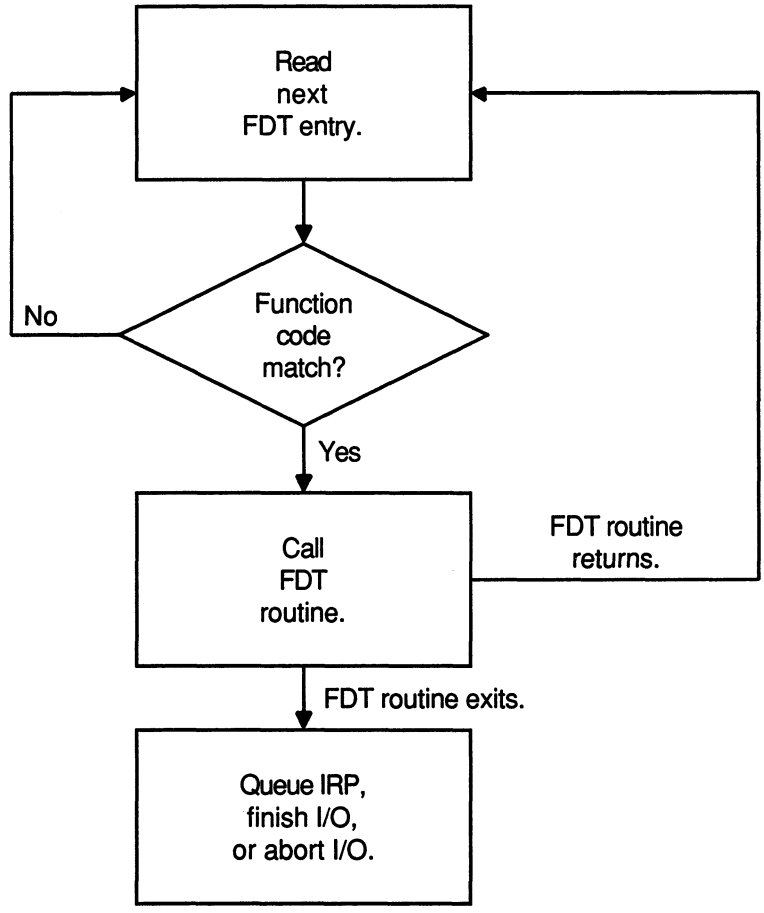
```
JSB     (R0)
```

Each FDT routine chooses an exit path based on the following factors:

- Whether another FDT routine needs to be called to perform additional function-specific processing
- Whether an error is found in the I/O request
- Whether the operation is complete
- Whether the I/O operation requires and is ready for device activity

The FDT routines, as illustrated in Figure 7–1, must transfer control out of the FDT processing loop and into a VMS routine that queues an IRP, completes an I/O request, or aborts an I/O request. The $QIO system service does not stop scanning the FDT. Therefore, you must ensure that for each valid function code in a driver's FDT, there is an FDT routine that does not return control to the $QIO system service.

Figure 7–1 $QIO Scan of a Function Decision Table



7.2.1 FDT Exit Paths

An FDT routine can exit using any of the following methods:

- RSB
- JMP G^EXE$QIODRVPKT
- JSB G^EXE$ALTQUEPKT
- JMP G^EXE$FINISHIO or JMP G^EXE$FINISHIOC
- JMP G^EXE$ABORTIO

These methods are described in the following sections, and you can find additional details on the routines they involve in the routines chapter of the *VMS Device Support Reference Manual*.

7.2.1.1 RSB

An FDT routine issues an RSB instruction to return to the $QIO system service. The FDT routine returns to the system service because the routine knows that the FDT contains a subsequent entry with the same function code bit set. As a result, the system service searches for another FDT routine.

7.2.1.2 JMP G^EXE$QIODRVPKT

EXE$QIODRVPKT transfers control to a VMS routine (EXE$INSIOQ) that either delivers an IRP immediately to a driver's start-I/O routine or places the IRP in a pending-I/O queue waiting for driver servicing. The FDT routine uses this exit method if all preprocessing is complete, if no fatal errors are found in the specification of an I/O request, and if device activity, synchronized access to the device's UCB, or synchronized access to device registers is required to complete the I/O request. Common examples of such a request are read and write functions.

EXE$INSIOQ transfers control to the device driver's start-I/O routine only if the device unit is currently idle. If the device unit is busy, EXE$INSIOQ inserts the IRP in a priority-ordered queue of IRPs waiting for the unit.

Once an FDT routine transfers control to EXE$QIODRVPKT, no driver code that further processes the I/O request can refer to process virtual address space. When a device driver's start-I/O routine gains control, the process that queued the I/O request might no longer be the mapped process. Therefore, the driver must assume that all information regarding the I/O request is in the UCB or the IRP and that all buffer addresses in the UCB are either system addresses or page-frame numbers that can be interpreted in any process context.

For direct I/O operations, FDT routines also must have locked all user buffer pages in physical memory because paging cannot occur at driver fork level or higher interrupt priority levels. The process virtual address space is not guaranteed to be mapped again until VMS delivers a special kernel-mode AST to the requesting process as part of I/O postprocessing.

7.2.1.3 JMP G^EXE$FINISHIO or JMP G^EXE$FINISHIOC

EXE$FINISHIO and EXE$FINISHIOC transfer control to a VMS routine that writes a quadword of final I/O status from R0 and R1 into the I/O status field of the IRP (IRP$L_MEDIA and IRP$L_MEDIA+4). (Note that EXE$FINISHIOC clears the second longword of the final I/O status.) The routine then inserts the IRP in the I/O postprocessing queue. These routines return to the $QIO system service the two longwords of status contained in the I/O status block (if any) specified in the I/O request.

An FDT routine that discovers a device-dependent error should always return status using EXE$FINISHIO or EXE$FINISHIOC. These routines gain control without any change in process context. Interrupt priority level is at IPL$_ASTDEL; the process page-tables are mapped; and the process is executing in kernel mode.

7.2.1.4 JMP G^EXE$ABORTIO

EXE$ABORTIO transfers control to a VMS routine that aborts an I/O request. An FDT routine that discovers a device-independent error should always use this method of exiting. Inability to gain access to a data buffer or an error in the specification of the I/O request are examples of device-independent errors.

EXE$ABORTIO gains control without any change in the process context. Interrupt priority level is at IPL$_ASTDEL; the process virtual space is mapped; and the process is executing in kernel mode. EXE$ABORTIO stores a longword of status in R0 and returns this to the system service.

7.2.1.5 JSB G^EXE$ALTQUEPKT

EXE$ALTQUEPKT transfers control to a VMS routine that calls an alternate start-I/O routine in the driver (specified in the driver dispatch table at offset DDT$L_ALTSTART) that synchronizes requests for activity on a device unit and initiates the processing of I/O requests.

The FDT routine uses this exit method when it has successfully completed all driver preprocessing and the request requires device activity. However, in contrast to EXE$QIODRVPKT, EXE$ALTQUEPKT bypasses the device unit's pending-I/O queue and the device busy flag; thus, the driver is activated regardless of whether the device unit is busy. A driver that can handle two or more I/O requests simultaneously uses this exit method.

Be aware that programming a device driver to process simultaneous I/O requests requires detailed knowledge of VMS internal design. A driver that uses EXE$ALTQUEPKT must not only maintain its internal queues but must also synchronize those queues with the unit's pending-I/O queue, which the operating system maintains. In addition, if a driver processes more than one IRP at the same time, it must use separate fork blocks. Such a driver completes the processing of I/O requests by calling the routine COM$POST. This routine places each IRP in the systemwide I/O postprocessing queue and returns control to the driver. The driver can then fetch another IRP from an internal queue. For more information about COM$POST, see the routines chapter of the *VMS Device Support Reference Manual*.

Unlike the other FDT exit routines, EXE$ALTQUEPKT is called with a JSB instruction rather than a JMP instruction. When the alternate start-I/O routine finishes, it returns control to EXE$ALTQUEPKT by executing an RSB instruction. The FDT routine performs any postprocessing and transfers control to the routine EXE$QIORETURN. When EXE$QIORETURN gains control, it performs the following steps:

1. Sets the success status code SS$_NORMAL in R0
2. Lowers the interrupt priority level to zero
3. Returns (with the RET instruction) to the system service dispatcher

7.3 FDT Routines for VMS Direct I/O

The VMS operating system provides two standard FDT routines that are applicable for direct I/O operations: EXE$READ and EXE$WRITE. When called by the driver, these routines completely prepare a direct I/O read or write request. Thus, a driver that uses these routines eliminates the need for its own device-specific FDT routines.

EXE$READ and EXE$WRITE are described in Section 7.5.

7.4 FDT Routines for VMS Buffered I/O

Device drivers for buffered I/O operations generally contain their own device-specific FDT routines.

An FDT routine for a buffered I/O data transfer operation should confirm either read or write access to the user's buffer and allocate a buffer in system space. Sections 7.4.1 and 7.4.2 describe these tasks.

An FDT routine for a buffered I/O operation that does not involve data transfer should copy the function-dependent parameters of the $QIO request (**p1** to **p6**) to the IRP, perform any necessary preprocessing, and use one of the exit methods listed in Section 7.2.1.

7.4.1 Checking Accessibility of the User's Buffer

First the FDT routine calls EXE$READCHK or EXE$WRITECHK to confirm write or read access, respectively, to the user's buffer. Both of these routines write the transfer byte count into IRP$L_BCNT. EXE$READCHK also sets IRP$V_FUNC in IRP$W_STS to indicate that the function is a read.

7.4.2 Allocating the System Buffer

Next, the FDT routine allocates a system buffer in the following manner:

1. It adds 12 bytes to the byte count passed in the **p2** argument of the user's I/O request, thus accommodating the standard size of a VMS buffer header. This is the total system buffer size.
2. It calls EXE$DEBIT_BYTCNT_ALO to ensure that the process's job has sufficient remaining byte count quota to allow its use of the requested buffer. If the job has sufficient quota, EXE$DEBIT_BYTCNT_ALO allocates the requested buffer from nonpaged pool, writes the buffer's size and type into its third longword, and subtracts the system buffer size from JIB$L_BYTCNT.

VMS also supplies the routines EXE$DEBIT_BYTCNT_BYTLM_ALO, EXE$DEBIT_BYTCNT(_NW), EXE$DEBIT_BYTCNT_BYTLM(_NW), and EXE$ALLOCBUF, which perform the same type of work as EXE$DEBIT_BYTCNT_ALO. These routines are fully described in the *VMS Device Support Reference Manual*.

Once the buffer is allocated, the FDT routine takes the following steps:

1 Loads the address of the system buffer into IRP$L_SVAPTE.

2 Loads the total size of the system buffer into IRP$W_BOFF.

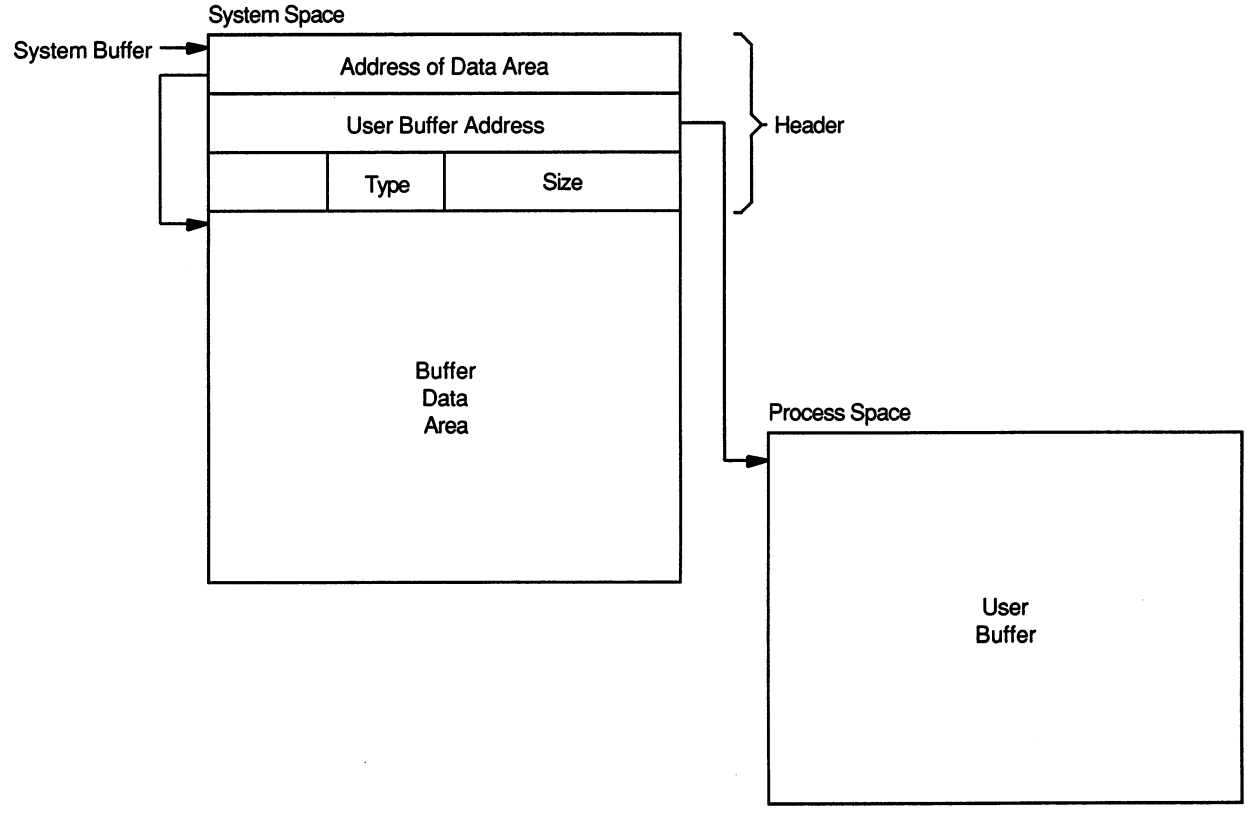
3 Stores the starting address of the system buffer data area in the first longword of the buffer header.

4 Stores the user's buffer address in the second longword of the header.

5 Copies data from the user buffer to the system buffer if the I/O request is a write operation.

At this point, the buffers are ready for the transfer. Figure 7–2 illustrates the format of the system buffer.

Figure 7–2 Format of System Buffer for a Buffered-I/O Read Function



7.4.3 Buffered-I/O Postprocessing

When the transfer finishes, the driver returns control to VMS for completion of the I/O request. The driver writes the final request status in the low-order word of R0. Use of the high-order word of R0 and the longword of R1 is driver specific. Certain drivers use these fields to report a transfer byte count, for example.

The driver must leave the buffer header intact; I/O postprocessing relies on the header's accuracy. When VMS I/O postprocessing gains control, it performs three steps:

1 Calls EXE$CREDIT_BYTCNT to add the value in IRP$W_BOFF to JIB$L_BYTCNT, thus updating the user's byte count quota

2 If IRP$L_SVAPTE is nonzero, assumes a system buffer was allocated and checks to see whether IRP$V_FUNC is set in IRP$W_STS

3 If IRP$V_FUNC is clear, deallocates the system buffer used for the write operation; if IRP$V_FUNC is set, the special kernel-mode AST copies the data to the user's buffer and then deallocates the buffer in addition to performing other kernel-mode AST functions

The special kernel-mode AST performs the following steps to complete a buffered read operation:

1 Obtains the address of the system buffer from IRP$L_SVAPTE.

2 Obtains the number of bytes to write to the user's buffer from IRP$L_BCNT.

3 Obtains the address of the user's buffer from the second longword of the system buffer header.

4 Checks for write accessibility on all pages of the user's buffer.

5 Copies the data from the system buffer to the process' buffer.

6 Deallocates the system buffer. Note that the system uses the size listed in the buffer's header to deallocate the buffer.

7.5 FDT Routines Provided by VMS

The VMS FDT routines perform I/O request validation that is common to many devices. Whenever possible, drivers should take advantage of these routines. Normally, if a VMS FDT routine is called, no additional FDT processing is required. All of the VMS FDT routines listed in Table 7–2 exit by transferring control to EXE$QIODRVPKT, EXE$FINISHIO, EXE$FINISHIOC, or EXE$ABORTIO. Once a VMS FDT routine is called, no subsequent FDT processing occurs.

For additional information about VMS FDT routines, see the pertinent routine descriptions in *VMS Device Support Reference Manual*.

Table 7–2 FDT Routines Provided by VMS

FDT Routine	Function	Exit Method
EXE$MODIFY	Processes a logical-read/write or physical-read/write function for a read and write direct I/O operation to a user-specified buffer	Aborts the I/O request if an error occurs, or dismisses the I/O request if the user I/O buffers cannot be locked in memory; otherwise, transfers control to EXE$QIODRVPKT
EXE$ONEPARM	Processes a nontransfer I/O function code that has one parameter associated with it	Transfers control to EXE$QIODRVPKT
EXE$READ	Processes a logical-read or physical-read function for a direct I/O operation	Aborts the I/O request if an error occurs, or dismisses and resubmits the I/O request if the user I/O buffers cannot be locked in memory; otherwise, transfers control to EXE$QIODRVPKT
EXE$SENSEMODE	Processes the sense-device-mode and sense-device-characteristics functions by reading fields of the UCB	Transfers control to EXE$FINISHIO
EXE$SETCHAR[1]	Processes the set-device-mode and set-device-characteristics functions	Transfers control to EXE$FINISHIO
EXE$SETMODE[1]	Processes the set-device-mode and set-device-characteristics functions by creating a driver fork process	Aborts the I/O request if an error occurs; otherwise, transfers control to EXE$QIODRVPKT
EXE$WRITE	Processes a logical-write or physical-write function for a direct I/O operation	Aborts the I/O request if an error occurs, or dismisses the I/O request if the user I/O buffers cannot be locked in memory; otherwise, transfers control to EXE$QIODRVPKT
EXE$ZEROPARM	Processes a nontransfer I/O function code that has no associated parameters	Transfers control to EXE$QIODRVPKT

[1] If setting device characteristics requires no device activity or requires no synchronization with fork processing, the driver's FDT entry can specify EXE$SETCHAR; otherwise, it must specify EXE$SETMODE.

8 Writing a Start-I/O Routine

A driver start-I/O routine activates a device and then waits for a device interrupt or timeout. This chapter describes the start-I/O routine. Chapter 10 describes the reactivation of the driver routine that performs device-dependent I/O postprocessing. With a few exceptions, the start-I/O routine discussed in the following sections describes a DMA transfer using a single-unit controller.

8.1 Transferring Control to the Start-I/O Routine

The start-I/O routine of a device driver gains control from either of two VMS routines: EXE$QIODRVPKT or IOC$REQCOM.

When FDT processing is complete for an I/O request, the FDT routine transfers control to EXE$QIODRVPKT which, in turn, calls EXE$INSIOQ. If the designated device is idle, EXE$INSIOQ calls IOC$INITIATE to create a driver fork process. (This procedure is detailed in Section 7.2.1.2.) The driver fork process then gains control in the start-I/O routine of the appropriate driver. If the device is busy, EXE$INSIOQ queues the packet to the device unit's pending-I/O queue.

After a device completes an I/O operation, the driver fork process exits by transferring control to IOC$REQCOM. IOC$REQCOM inserts the IRP for the finished transfer into the postprocessing queue. It then dequeues the next IRP from the device unit's pending-I/O queue and calls IOC$INITIATE to initiate the processing of this I/O request in the driver's fork process at the entry point of the driver's start-I/O routine.

8.2 Context of a Driver Fork Process

A start-I/O routine does not run in the context of a user process. Rather, it has the following context:

System context	Driver code can only refer to system virtual addresses.
Kernel mode	Execution occurs in the most privileged access mode and can, therefore, change IPL and obtain spin locks.
High IPL	The VMS routine that creates a driver fork process obtains the driver's fork lock, raising IPL to driver fork level before activating the driver.
Kernel or interrupt stack	Execution occurs on the kernel or interrupt stack. The driver must not alter the state of the stack without restoring the stack to its previous state before relinquishing control. The stack used depends on whether the I/O startup is the result of a new I/O request or because a previously requested I/O operation has been completed. The choice of stacks must not affect the operation of the start-I/O routine.

In addition to the context described, the VMS packet-queuing routines set up R3 and R5 for a driver start-I/O routine, as follows:

- R3 contains the address of the IRP.
- R5 contains the address of the UCB for the device.

The start-I/O routine must preserve all general registers except R0, R1, R2, and R4.

Before the packet-queuing routines call the start-I/O routine, they copy the following IRP fields into their corresponding slots in the device's UCB:

- IRP$L_BCNT (low-order word) → UCB$W_BCNT
- IRP$W_BOFF → UCB$W_BOFF
- IRP$L_SVAPTE → UCB$L_SVAPTE

8.3 Functions of a Start-I/O Routine

The processing performed by a start-I/O routine is device specific. A start-I/O routine normally contains elements that perform the following functions to activate:

- Analyzing the I/O function
- Transferring the details of a request from the IRP into the UCB
- Obtaining and initializing the controller
- Modifying device registers to activate the device

A start-I/O routine of a DMA device driver performs additional tasks to prepare the device for a DMA transfer prior to activating the device. These tasks include the following:

- Obtaining I/O adapter resources such as map registers and a buffered data path
- Computing the starting address of a data transfer

The following sections describe the general activities of a start-I/O routine for a typical device. The details of DMA processing are specific to the particular device. Section 14.2 describes the UNIBUS- and Q22 bus-related details of DMA transfers. Section 15.5.3 relates those tasks that MASSBUS DMA device drivers must perform. Section 16.6 discusses similar functions that drivers for generic VAXBI devices may need to perform.

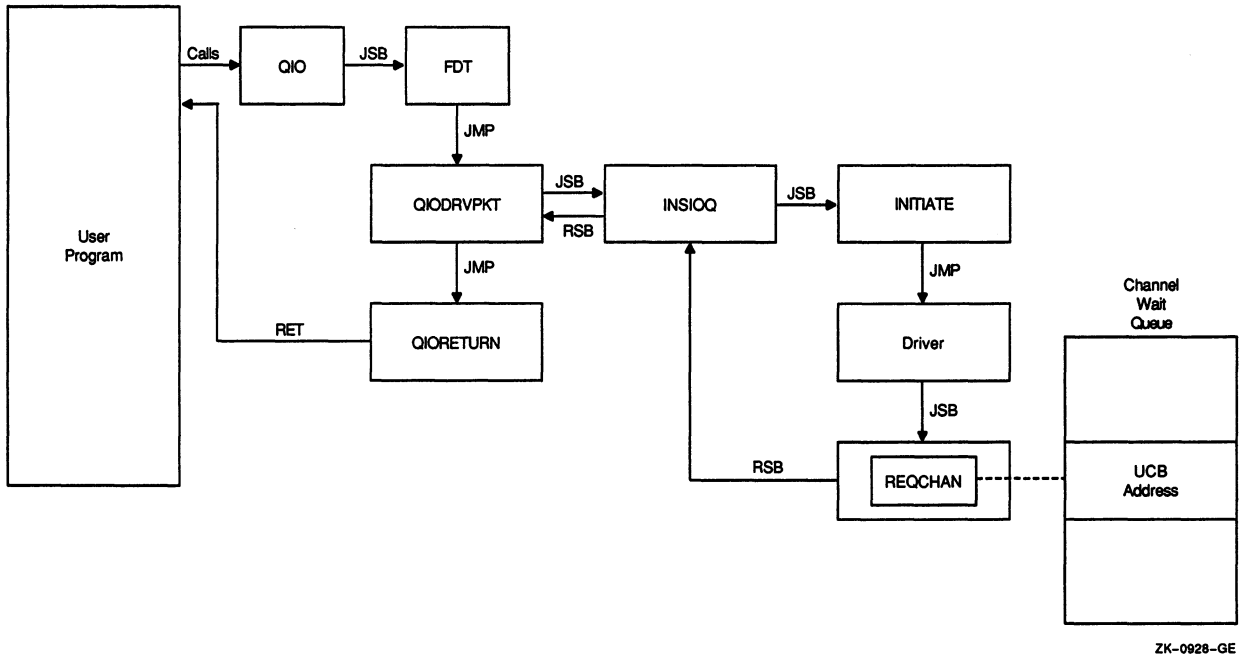
8.3.1 Obtaining Controller Access

If the device is one of several attached to a controller, the start-I/O routine invokes the VMS macro REQPCHAN to assign the controller's data channel to the device unit. Controllers that control only one device do not require arbitration for the controller's data channel. REQPCHAN calls the VMS routine IOC$REQPCHANL that acquires ownership of the controller data channel.

The transfer being controlled by the start-I/O routine discussed here requires no seek preceding the transfer. Disk I/O is an example of a transfer that requires a seek first. To permit seeks to be overlapped with transfers, invoke REQPCHAN with the argument **pri**=HIGH. Specifying **pri**=HIGH inserts a request for a channel at the head of the channel wait queue.

If the channel is not available, IOC$REQPCHANL suspends driver processing by saving the driver's context in the UCB fork block and inserting the fork block in the channel wait queue. IOC$REQPCHANL then returns control to the caller of the driver, that is, to EXE$INSIOQ, as illustrated in Figure 8–1. This procedure is further discussed in Section 3.4.1.

Figure 8–1 Inserting a UCB into the Channel Wait Queue



The UCB fork block now represents the entire context of the suspended driver:

- Saved R3 containing the IRP address
- Implicitly saved R5 containing the UCB address
- A return address in the driver

Note that, because IOC$RELCHAN moves the address of the device's CSR into R4 before resuming a suspended driver, IOC$REQPCHANL does not save R4 in the UCB fork block.

If the channel is available, IOC$REQPCHANL locates the interrupt dispatch block (IDB) for the channel with a pointer in the UCB:

UCB → CRB → IDB

The IDB contains the address of the control and status register (CSR) for the channel (IDB$L_CSR). IOC$REQPCHANL returns the CSR address in R4. The driver for a unit attached to a dedicated controller must contain the code needed to load the CSR address into R4.

IOC$REQPCHANL also writes the address of the new channel-owner's UCB in the owner field of the IDB (IDB$L_OWNER). The driver's interrupt service routine later reads this IDB field to determine which device unit owns the controller's data channel. A driver for a single-unit controller must fill the IDB$L_OWNER field in its controller or unit initialization routines.

The driver must maintain the stack in a known and consistent state for the resource-wait-queue mechanism to work. When IOC$REQPCHANL gains control, the top two items on the stack must be two return addresses:

- 00(SP)—Address of the next instruction to be executed in the driver fork process. The transfer of control to IOC$REQPCHANL places this address on the stack.
- 04(SP)—Address of the next instruction to be executed in the routine that called the driver start-I/O routine.

8.3.2 Obtaining and Converting the I/O Function Code and Its Modifiers

The start-I/O routine extracts the I/O function code and function modifiers from the field IRP$W_FUNC and translates them into device-specific function codes, which it loads into the device's CSR or other control registers. The start-I/O routine creates and modifies a bit mask that is to be loaded into the CSR when the driver starts the device. To accomplish this, the start-I/O routine converts the function modifiers contained in IRP$W_FUNC into device-specific bit settings in the general register.

At this point, a UNIBUS/Q22 bus DMA driver follows procedures to obtain I/O bus resources and compute the size and starting address of a transfer. These procedures are discussed in Section 14.2. MASSBUS DMA device drivers perform the steps indicated in Section 15.5.3.

8.3.3 Preparing the Device Activation Bit Mask

For a typical device, the start-I/O routine prepares the device-activation bit mask by setting the interrupt-enable bit and the go bit in the general purpose register that also contains the high-order bits of the bus address and the device-function bits. At this point, the general register contains a complete command for starting the transfer, also known as the **control mask**.

When the start-I/O routine copies the contents of the register into the device's CSR, the device starts the transfer. Before activating the device, however, the start-I/O routine should perform the steps described in Sections 8.3.4 and 8.3.5.

8.3.4 Synchronizing Access to the Device Database

The start-I/O routine invokes the VMS macro DEVICELOCK to synchronize its access to device registers with the interrupt service routine. This macro invocation is doubly important, for it establishes the context wherein the driver can later issue the wait-for-interrupt macro (WFIKPCH or WFIRLCH). The wait-for-interrupt macros expect the driver's fork IPL to be on the stack, as placed there by the DEVICELOCK macro. In addition, the wait-for-interrupt macros issue the DEVICEUNLOCK macro to release ownership of the device lock and restore the previous IPL.

8.3.5 Checking for a Local Processor Power Failure

After synchronizing access to device registers, the start-I/O routine invokes the VMS macro SETIPL to raise IPL to IPL$_POWER to block all interrupts on the local processor.

The start-I/O routine then examines the powerfail bit in the UCB's status longword (UCB$V_POWER in UCB$L_STS) to determine whether a local power failure has occurred since the start-I/O routine gained control. If the bit is not set, the transfer can proceed.

If the bit is set, a power failure might have occurred between the time that the start-I/O routine wrote the first device register and the time that the start-I/O routine is ready to activate the device. Such a power failure could modify the already-written device registers and cause unpredictable device behavior if the device were to be started.

If the bit UCB$V_POWER is set, the start-I/O routine branches to an error handler in the driver. The driver error handler must perform the following actions:

- Clear UCB$V_POWER
- Issue the DEVICEUNLOCK macro to release the device lock and restore IPL to fork IPL

After performing these tasks, many drivers transfer control to the beginning of the start-I/O routine, which restarts the processing of the I/O request.

8.3.6 Activating the Device

If no power failure has occurred, the start-I/O routine copies the contents of the control mask into the device's CSR. When the device notices the new contents of the device register, it begins to transfer the requested data.

8.4 Waiting for an Interrupt or Timeout

Once the start-I/O routine activates the device, the driver fork process cannot proceed until one of these events occurs:

- The device generates a hardware interrupt.
- The device does not generate a hardware interrupt within an expected time limit, which is to say that a device timeout occurs.

Still executing at IPL$_POWER, the driver's start-I/O routine asks VMS to suspend the driver fork process by invoking one of the following macros:

WFIKPCH	Wait for an interrupt or timeout and keep the controller data channel
WFIRLCH	Wait for an interrupt or timeout and release the controller data channel

The WFIKPCH and WFIRLCH macros require the address of a timeout handling routine in the **excpt** argument. Optionally, but almost always, the driver can also indicate the number of seconds the system must wait before signaling a timeout in the **time** argument. A full description of these macros appears in the macro chapter of the *VMS Device Support Reference Manual*.

Both macros invoke routines that release ownership of the device lock, relinquish synchronization, and return IPL to the previous level when exiting. These routines expect to find the return IPL on the stack. This IPL is saved on the stack by the DEVICELOCK macro as described in Section 8.3.4.

Drivers generally keep the controller data channel while waiting for the interrupt or timeout. Drivers of devices with dedicated controllers always keep the channel because only one unit ever needs it. For devices that share a controller, some operations, such as disk seeks, do not require the controller once the operation has begun. In such cases, the driver can release the controller's data channel while waiting for an interrupt or timeout so that other units on the controller can start their operations.

8.4.1 Expansion of WFIKPCH Macro

Because the WFIKPCH and WFIRLCH macros are similar, the description that follows analyzes the expansion of WFIKPCH only.

If the driver specifies the **time** argument in the macro call, the macro pushes the value of the argument into the stack. If the **time** argument is not specified, the macro pushes the value 65,536 onto the stack. IOC$WFIKPCH uses the time value to calculate the length of time VMS waits before transferring control to a device timeout handler.

WFIKPCH completes its expansion with two lines of code:

```
JSB     G^IOC$WFIKPCH
.WORD   EXCPT-.
```

The execution of the JSB instruction pushes the address following the JSB onto the stack as the address to which the called routine would normally return with an RSB instruction.

8.4.2 IOC$WFIKPCH Routine

The VMS routine IOC$WFIKPCH, invoked by the macro WFIKPCH, performs the functions necessary for the driver fork process to wait for a device interrupt or timeout. IOC$WFIKPCH first adds 2 to the address on the top of the stack so that the top of the stack contains the address of the next instruction in the driver after the macro invocation. This address is where the driver resumes execution as a result of an interrupt service routine's JSB instruction.

IOC$WFIKPCH then saves the contents of R3, R4, and the address to which control must be returned to the driver, which it takes from the top of the stack. It saves this information in the first part of the UCB in the UCB fork block.

Note that, after an interrupt, the interrupt service routine must restore R5 so that it contains the address of the UCB. The interrupt service routine normally obtains the address of the UCB from the field IDB$L_OWNER of the IDB.

The VMS routine that detects a device timeout calculates the address of the driver's timeout routine by subtracting 2 from the saved PC in the UCB's fork block and calling indirectly through the result. For example:

```
MOVL    UCB$L_FPC(R5),R2        ; Get saved PC
CVTWL   -(R2),-(SP)             ; Get offset to timeout
                                ; handler
ADDL    (SP)+,R2                ; Add to relative driver
                                ; address to obtain relative
                                ; handler address
JSB     (R2)                    ; Call timeout handler
```

IOC$WFIKPCH sets bits in the UCB (UCB$V_INT and UCB$V_TIM in UCB$L_STS) to indicate that interrupts and timeouts are expected from the device. IOC$WFIKPCH also writes the device timeout absolute time in the field UCB$L_DUETIM. The absolute time is the number of seconds since the operating system was bootstrapped plus the number of seconds specified in the **time** argument to the macro.

Finally, IOC$WFIKPCH reenables interrupts by releasing the device lock and lowering IPL to fork level, the IPL at which the driver was executing previously. It then returns control to the caller of the driver.

9 Writing an Interrupt Service Routine

When a device generates a hardware interrupt, it requests an interrupt at the appropriate device IPL. Either the device or its adapter requests a processor interrupt at that IPL. When the processor executes at an IPL below that device IPL, interrupt dispatching begins.

The mechanism of interrupt dispatching has no direct bearing on the contents of a driver's interrupt service routine. Its implementation varies slightly according to the VAX processing system and I/O subsystem in use. To obtain background information on the dispatcher, refer to the overview provided in Section 14.3, which also details the method of dispatching UNIBUS/Q22 bus device interrupts. MASSBUS device driver writers should refer also to Section 15.4; generic VAXBI device driver writers should read the discussion in Section 16.4.1.

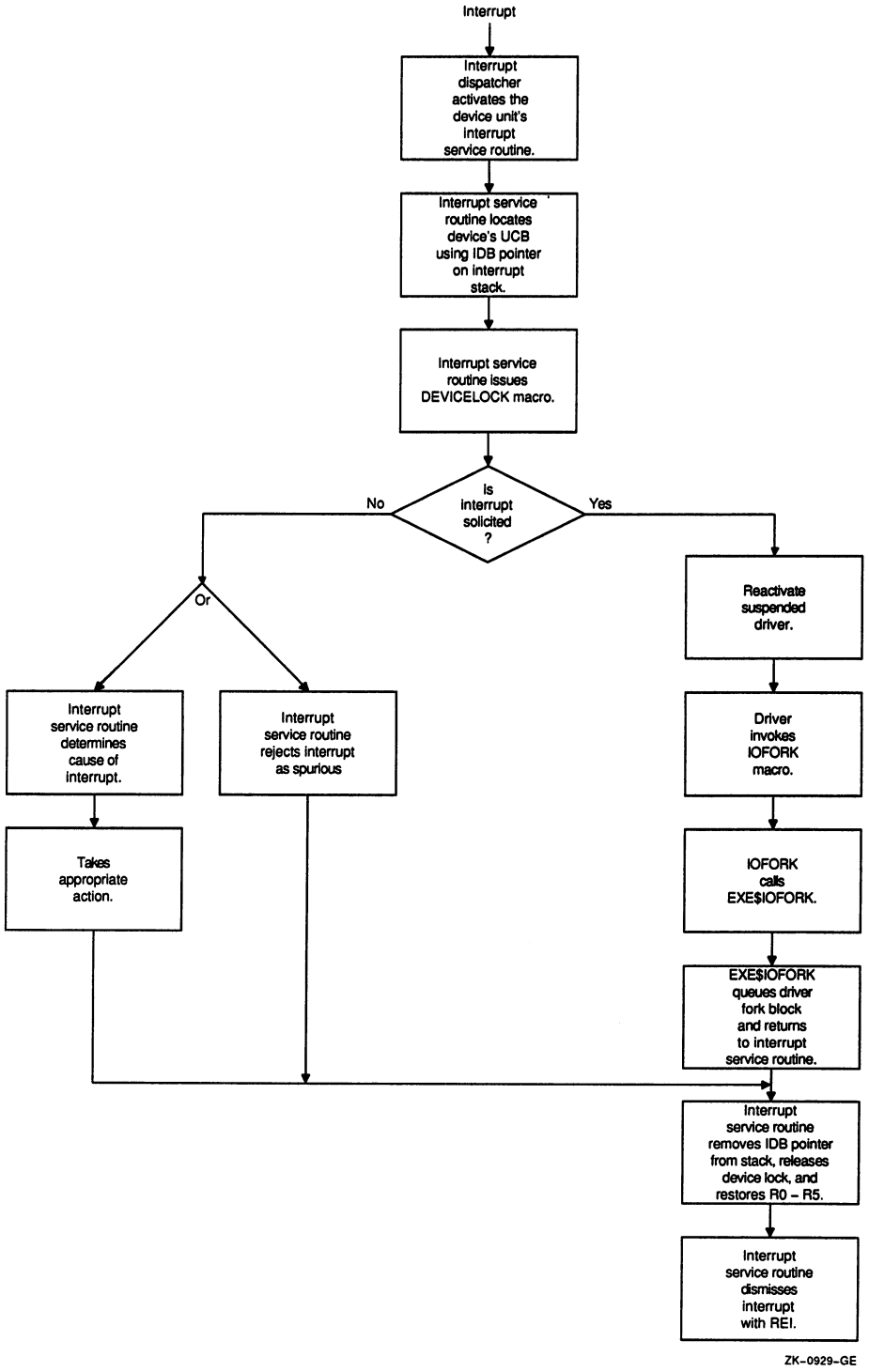
For most device drivers, the driver prologue table contains, in the reinitialization section established by the DPT_STORE macro, the address of one or more interrupt service routines. Each interrupt service routine corresponds to an interrupt vector on the I/O bus. You specify the address of an I/O bus vector using the SYSGEN command CONNECT, as described in Section 12.2.2.

Most device interrupt service routines perform the following functions:

- Locate the device's UCB
- Determine whether the interrupt was solicited
- Reject or process unsolicited interrupts
- Activate the suspended driver to process solicited interrupts

Figure 9–1 illustrates the general flow of interrupt handling. The remaining sections of this chapter describe the handling of solicited and unsolicited interrupts in further detail.

Figure 9–1 Flow of Interrupt Servicing



9.1 Interrupt Context

When the interrupt dispatcher calls a driver's interrupt service routine, execution context is as follows:

- R0 through R5 are saved on the stack.
- Only system address space may be accessed.
- IPL is at hardware device interrupt level.
- The processor is running in kernel mode.
- The processor is running on the interrupt stack.

The stack contains the following information:

Stack Location	Content
00(SP)	Pointer to the address of the IDB
04(SP) through 24(SP)	Saved R0 through R5
28(SP)	PC at the time of the interrupt
32(SP)	PSL at the time of the interrupt

In the course of its processing, an interrupt service routine must remove the IDB pointer and the saved registers from the stack before dismissing the interrupt with an REI instruction.

9.2 Servicing a Solicited Interrupt

When a driver's fork process activates a device and expects to service a device interrupt as a result, the fork process suspends its execution and waits for an interrupt to occur. The suspended driver is represented only by the contents of the fork block in the device's UCB and the stack, which contain the following information:

- A description of the I/O request and the state of the device
- The contents of R3 and R4
- The implicit contents of R5 (the address of the UCB fork block)
- The address at which to return control to the driver
- The implicit address of a timeout handling routine

When the interrupt service routine returns control to the main line of driver processing, it has only restored the contents of R3, R4, R5, and the PC.

A driver's interrupt service routine performs the following tasks to process the interrupt and transfer control to the waiting driver:

1 Obtains the address of the device's UCB from the IDB, as follows:

$$00(SP) \rightarrow CRB \rightarrow IDB \rightarrow IDB\$L\_OWNER \rightarrow UCB$$

The interrupt service routine restores the UCB address to R5.

2 Issues the DEVICELOCK macro to obtain synchronized access to device registers.

3 Tests the interrupt-expected bit in the UCB status longword (UCB$V_INT in UCB$L_STS). If the bit is set, the driver is waiting for an interrupt from this device. After performing this test, the interrupt service routine *must* clear UCB$V_INT to indicate that it has received the expected interrupt.

Note: Because device timeout processing mostly occurs at fork IPL (see Section 10.2), a driver's interrupt service routine, executing at device IPL, could interrupt the processing of a timeout on the same device unit. For this reason, the driver's interrupt service routine should check the interrupt-expected bit (UCB$V_INT) before handling the interrupt. VMS clears this bit before it calls the driver's timeout handler.

4 Obtains device-status or controller-status information from the device registers, if necessary, and stores the status information in the UCB.

5 Places the contents of UCB$L_FR3 and UCB$L_FR4 in R3 and R4, respectively.

6 Issues a JSB instruction to the waiting driver's PC address, which is saved in the UCB fork block at UCB$L_FPC.

The restored driver should execute as briefly as possible in interrupt context. As soon as possible, the driver should invoke the IOFORK macro to request the creation of a fork process at the driver's fork IPL. It must do this in order to complete the I/O operation. Forking lowers the IPL of driver execution below device IPL, allowing the processor to service additional device interrupts. IOFORK calls the routine EXE$IOFORK. EXE$IOFORK inserts into the appropriate fork queue the UCB fork block that describes the driver process. It then returns control to the driver's interrupt service routine. (See Section 10.1.1 for additional information on driver forking.)

The interrupt service routine then performs the following steps:

1 Removes the IDB pointer from the stack

2 Issues the DEVICEUNLOCK macro to release ownership of the device lock

3 Restores R0 through R5

4 Dismisses the interrupt with an REI instruction

9.3 Servicing an Unsolicited Interrupt

A device requests an interrupt to indicate to a driver that the device has changed status. If a driver's fork process starts an I/O operation on a device, the driver expects to receive an interrupt from the device when the I/O operation completes or an error occurs.

Other changes in the device's status occur when the device has not been activated by a device driver. The device reports such a change by requesting an unsolicited interrupt. For example, when a user types on a terminal, the terminal requests an interrupt that is handled by the terminal driver. If the terminal is not attached to a process, the terminal driver causes the login procedure to be invoked for the user at the terminal.

As another example, an unsolicited interrupt occurs whenever a disk drive goes off line, as could happen when an operator spins it down or pushes the offline button. The disk driver services the interrupt by altering volume and unit status bits in the disk device's UCB.

Devices request unsolicited interrupts because some external event has changed the status of the device. A device driver can handle these interrupts in two ways:

- Ignore the interrupt as spurious
- Examine the device registers and take action according to their indications of changed status, and then poll for any other changes in device status

As mentioned in Section 9.2, an interrupt service routine first obtains the address of the device's UCB from the IDB. It then issues the DEVICELOCK macro to obtain synchronized access to device registers.

The routine determines whether an interrupt is solicited or not by examining the interrupt-expected bit in the UCB status longword (UCB$V_INT in UCB$L_STS). All UNIBUS, Q22 bus, and generic VAXBI device drivers must use this method to determine whether or not an interrupt is solicited; the address of the unsolicited interrupt service routine, specified in the driver dispatch table, is used only by MASSBUS drivers (see Sections 15.4 and 15.6).

If the interrupt is unsolicited, the driver can reject the interrupt with the following code sequence:

1 Remove the IDB pointer from the stack

2 Restore R0 through R5

3 Dismiss the interrupt with an REI instruction

If the driver decides to handle the unsolicited interrupt, it must observe certain precautions. Certain methods of servicing unsolicited interrupts—for instance sending a message to the operator or the job controller's mailbox—must be accomplished at an IPL lower than device IPL. Although the interrupt service routine can legitimately fork to accommodate unsolicited interrupts, it should exercise extreme caution in doing so.

If UCB$V_BSY is set in UCB$L_STS, the UCB fork block is currently in use by the driver's start-I/O routine. An attempt by the interrupt service routine to concurrently use the fork block can destroy the fork context already stored in that UCB. Moreover, if UCB$V_BSY is not set, the interrupt service routine cannot safely assume that the fork block is not in use, for it may be currently employed to service a previous unsolicited interrupt.

To avoid confusion, code servicing an unsolicited interrupt must ensure that the fork block it requires is not being used. Perhaps the safest method to guarantee this is for the driver to define a separate fork block in a device-specific UCB extension. The driver should also define a semaphore bit to control access to this fork block and protect against multiple forking. Note that the driver should access the semaphore bit using interlocked instructions (for example, BBSSI or BBCCI).

If, upon servicing an unsolicited interrupt, the driver's interrupt service routine examines the semaphore and discovers that a fork is already in progress (that is, the bit is set), it should not attempt to fork.

The VMS routine that creates the fork process (once these conditions are satisfied) returns control to the interrupt service routine. The interrupt service routine then releases the device lock, restores the saved registers, and issues an REI instruction to dismiss the interrupt.

9.3.1 Examples of Unsolicited Interrupts

A card reader requests an unsolicited interrupt when a user puts the reader on line. Once the card-reader driver's interrupt service routine determines that the interrupt is unsolicited, the routine analyzes the interrupt, as in the following code example.

Because only one sequence of instructions can use the UCB as a fork block, the interrupt service routine performs the following steps before it creates the fork process:

- Ensures that no one is using the device, and that no one wants to use it, by determining that the reference count (UCB$W_REFC) is zero.
- Ensures that it is not already using the UCB, to create a fork process in order to lower IPL and to send a message to the job controller, by testing the job-attached bit (UCB$V_JOB in UCB$W_DEVSTS).

```
CR$INT::
        MOVL    @(SP)+,R3                       ;Get address of IDB❶
        MOVQ    IDB$L_CSR(R3),R4                ;Get controller CSR and owner UCB
                                                ; address❷
        DEVICELOCK  LOCKADDR=UCB$L_DLCK(R5),-
                PRESERVE=NO,-
                CONDITION=NOSETIPL              ;Obtain device lock❸
        BBCC    #UCB$V_INT,UCB$L_STS(R5),10$ ;If clear, interrupt not expected❹
        .
        .
        .
;
; UNSOLICITED INTERRUPT
;
10$:    MOVZWL  CR_CSR(R4),R0                   ;Get reader status
        MOVZBW  #CR_CSR_M_IE,CR_CSR(R4)         ;Clear status, enable interrupts❺
        BITW    #CR_CSR_M_ONLINE,R0             ;Reader transition to online?❻
        BEQL    20$                             ;If equal no
        TSTW    UCB$W_REFC(R5)                  ;Device assigned or allocated?❼
        BNEQ    20$                             ;If not equal yes
        BBSS    #UCB$V_JOB,UCB$W_DEVSTS(R5),-
                20$                             ;If set, message already sent❽
        BSBB    30$                             ;Send message to job controller
```

```
20$:    DEVICEUNLOCK  LOCKADDR=UCB$L_DLCK(R5),-
                      PRESERVE=NO                   ;Release device lock
        MOVQ          (SP)+,R0                      ;Restore registers
        MOVQ          (SP)+,R2
        MOVQ          (SP)+,R4
        REI
30$:    FORK                                        ;Create fork process❾
        MOVZBL        #MSG$_CRUNSOLIC,R4            ;Set message type❿
        MOVL          G^SYS$AR_JOBCTLMB,R3          ;Set address of job controller mailbox
        JSB           G^EXE$SNDEVMSG                ;Sent message to job controller
        BLBS          R0,40$                        ;If LBS successful notification⓫
        BICW          #UCB$M_JOB,UCB$W_DEVSTS(R5)   ;Clear message sent bit⓬
40$:    RSB
```

❶ The interrupt service routine obtains the address of the IDB from the top of the stack.

❷ By means of this action, it obtains the address of the control and status register (CSR) in R4 and restores the address of the UCB in R5.[1]

❸ It issues a DEVICELOCK macro to secure synchronized access to device registers and UCB fields.

❹ It checks for an unsolicited interrupt by testing the interrupt expected bit in the UCB status longword.

❺ Because the interrupt is unsolicited, the routine clears all CSR bits except for the interrupt-expected bit.

❻ It confirms that the reader was just placed on line by examining a saved copy of the CSR.

❼ It examines the reference count field of the device's UCB (UCB$W_REFC) to determine whether a process has allocated the device or assigned a channel to it.

❽ If the reference count is zero, the interrupt service routine tests the job-attached bit in the device-dependent status field (UCB$V_JOB in UCB$W_DEVSTS) to make sure it has not already sent the job controller a message about the card reader being placed on line.

❾ If the job-attached bit is not set, the routine sets the bit and creates a fork process to send the message to the job controller, using the system routine EXE$SNDEVMSG (described in the *VMS Device Support Reference Manual*). It is necessary to lower IPL from device IPL by forking at this point because EXE$SNDEVMSG expects its caller's IPL to be no greater than IPL$_MAILBOX.

When the interrupt service routine regains control, it releases the device lock, restores R0 through R5 and dismisses the interrupt with an REI instruction. (The interrupt service routine removed the IDB pointer from the stack earlier in its execution in order to obtain CSR and UCB addresses.)

---

[1] Because the card reader has a dedicated controller, the IDB$L_OWNER field always points to the UCB for the single unit:

00(SP) → CRB → IDB → IDB$L_OWNER → UCB

⑩ When the fork process created at step 8 eventually executes, it writes a message to the job controller's mailbox, indicating that the card reader is on line.

⑪ If the fork process successfully sends the message, it leaves the job-attached bit set to prevent the job controller from receiving any further messages about the card reader's state. (The driver's cancel-I/O routine later clears the bit.)

⑫ If the send-message request fails, the fork process clears the job-attached bit so that if the card reader makes a subsequent state change to on line, the interrupt service routine can attempt again to send a message to the job controller.

Another example of unsolicited interrupt processing occurs in a device driver for a multiunit controller. When a disk is placed off line, the disk drive hardware requests an interrupt. The driver interrupt service routine must determine what device unit requested the interrupt, obtain status information from the disk device's CSR, and then decide whether the interrupt was solicited.

Because it must access device UCB fields and device registers, the interrupt service routine first obtains the appropriate device lock. If the interrupt is unexpected, it calls code that services the unsolicited interrupt. This code checks the status of the volume, as described in the following steps:

1 It sets a bit in the UCB to indicate that the unit is on line (UCB$V_ ONLINE in UCB$L_STS).

2 If the UCB's volume-valid bit is set (UCB$V_VALID in UCB$L_STS), the routine tests the volume valid status bit in a device register to determine whether the volume status has changed. If the volume is no longer valid, the routine clears the UCB volume valid bit.

3 The routine returns control to the driver's interrupt service routine.

The driver's interrupt service routine then polls the other device units on the controller to determine whether any other units requested interrupts while the first interrupt was being processed. When no unit requires interrupt servicing, the routine removes the IDB pointer from the stack, releases the device lock, restores registers R0 through R5, and dismisses the interrupt with an REI instruction.

10 Completing an I/O Request and Handling Timeouts

Once a driver has activated the device and invoked the wait-for-interrupt macro, the driver remains suspended until the device requests an interrupt or times out.

If the device requests an interrupt, the driver's interrupt service routine handles the interrupt and then reactivates the driver at the instruction following the wait-for-interrupt macro. The reactivated driver performs device-dependent I/O postprocessing.

If the device does not request an interrupt within the designated time interval, the system transfers control to the driver's timeout handling routine. The address of the timeout handling routine is specified as the **excpt** argument to the wait-for-interrupt macro.

10.1 I/O Postprocessing

Once the driver interrupt service routine has processed an interrupt, it transfers control to the driver by issuing a JSB instruction. At this point, the driver is executing in interrupt context. If the driver were to continue executing in interrupt context, it would lock out most other processing on the processor including the handling of hardware interrupts.

To restore the driver to the context of a driver fork process, the driver invokes the VMS macro IOFORK. Once the fork process has been created and dispatched for execution, it executes the driver code that completes the processing of the I/O request.

10.1.1 EXE$IOFORK

IOFORK generates a call to the routine EXE$IOFORK. EXE$IOFORK converts the driver context from that of an interrupt service routine to that of a fork process by performing the following steps:

1. It disables software timeouts by clearing the timeout enable bit in the UCB status longword (UCB$V_TIM in UCB$L_STS).
2. It saves R3 and R4 of the current driver context in the UCB fork block (UCB$L_FR3 and UCB$L_FR4).
3. It saves the current driver PC in the UCB fork block (UCB$L_FPC). (The driver PC is the top longword on the stack, as a result of the JSB to EXE$IOFORK.)
4. It obtains the fork lock index of the driver from the UCB (UCB$B_FLCK) and uses it to determine in which fork queue it should place the fork block.
5. It inserts the address of the UCB fork block (R5) into the processor-specific fork queue corresponding to the driver's fork IPL.

6 Finally, if the fork block is the first entry in the fork queue, EXE$IOFORK requests a software interrupt from the local processor at the driver's fork IPL.

The steps listed previously move the fork process's context into the UCB's fork block. They save R3 through R5 and the driver's PC address. The driver's fork process resumes processing when the VMS fork dispatcher dequeues the UCB fork block from the fork queue, and reactivates the driver at the driver's fork IPL.

10.1.2 Completing an I/O Request

When VMS reactivates a driver's fork process by dequeuing the fork block, the driver resumes processing of the I/O operation holding the appropriate fork lock at fork IPL. Generic VAXBI devices perform whatever device-dependent operations are needed to prepare an I/O request for completion. If the device has completed the I/O operation without errors, a UNIBUS or Q22 bus driver for a DMA device proceeds as follows:

1 Purges the data path

2 Releases the buffered data path (applies only to UNIBUS DMA device drivers)

3 Releases map registers

4 Releases the controller (applies only to drivers of devices on multiunit controllers)

5 Checks device register images saved in the UCB to determine the status of the I/O operation

6 Saves in the IRP the status code, transfer count, and device-dependent status that is to be returned to the user process in an I/O status block

7 Returns control to the operating system

The first three steps apply to UNIBUS or Q22 bus DMA transfers only and are discussed in Section 14.2. The following sections describe the last steps.

10.1.2.1 Releasing the Controller

To release the controller channel, the driver code invokes the VMS macro RELCHAN. RELCHAN calls the VMS routine IOC$RELCHAN. If another driver is waiting for the controller channel, IOC$RELCHAN grants that driver's fork process the channel, restores its context from the UCB fork block, and transfers control to the saved PC. When no more drivers are awaiting the channel, IOC$RELCHAN returns control to the fork process that released the channel.

Drivers for devices with dedicated controllers need not release the controller's data channel (as discussed in Sections 8.3.1 and 11.1). By means of code in the unit initialization routine, these drivers set up the device's UCB so that the device owns the controller permanently.

Drivers must be executing at driver's fork IPL when they invoke RELCHAN or call IOC$RELCHAN.

10.1.2.2 Saving Status, Count, and Device-Dependent Status

To save the status code, transfer count, and device-dependent status, the driver performs the following steps:

1. Loads a success status code (SS$_NORMAL), or whatever is appropriate, into bits 0 through 15 of R0.
2. Loads the number of bytes transferred into the high-order 16 bits of R0 (bits 16 through 31), if the I/O operation performed by the device is a transfer function.
3. Loads device-dependent status information, if any, into R1.[1]

10.1.2.3 Returning Control to the Operating System

Finally, the driver fork process returns control to the system by invoking the REQCOM macro to complete the I/O request. REQCOM issues a JMP instruction to the VMS routine IOC$REQCOM. IOC$REQCOM locates the address of the I/O request packet (IRP) corresponding to the I/O operation in the device's UCB (UCB$L_IRP). It then writes the two longwords of completion status contained in R0 and R1 into the media field of the IRP (IRP$L_MEDIA and IRP$L_MEDIA+4).

IOC$REQCOM then inserts the IRP in the local processor's I/O-postprocessing queue and requests a software interrupt at IPL$_IOPOST from the local processor so the postprocessing begins when IPL drops below IPL$_IOPOST.

If the error-logging bit is set in the device's UCB (UCB$V_ERLOGIP in UCB$L_STS), IOC$REQCOM obtains the address of the error message buffer from the UCB (UCB$L_EMB). It then writes the following information into the error buffer:

- Final device status (UCB$W_DEVSTS)
- Final error count (UCB$B_ERTCNT)
- Maximum error retry count for the driver
- Two longwords of completion status (R0 and R1)

To release the error message buffer, IOC$REQCOM calls ERL$RELEASEMB. Section 11.3 describes error logging in more detail.

If any IRPs are waiting for driver processing, IOC$REQCOM dequeues an IRP from the head of the queue of packets waiting for the device unit (UCB$L_IOQFL), and transfers control to IOC$INITIATE. IOC$INITIATE initiates execution of this I/O request in the driver's fork process, by activating the driver's start-I/O routine, as described in Section 4.2.1.

Otherwise, IOC$REQCOM clears the unit-busy bit in the device's UCB status longword (UCB$V_BSY in UCB$L_STS) and transfers control to IOC$RELCHAN to release the controller channel in case the driver failed to do so. IOC$RELCHAN, in turn, returns control to the caller of the driver fork process (if the fork process issued the REQCOM macro). This is generally the VMS fork dispatcher. The fork dispatcher releases the

[1] R0 and R1 are the status values that VMS returns to the user process in the I/O status block specified in the original $QIO system service.

fork lock, restores saved registers, and dismisses the fork IPL software interrupt with an REI instruction.

The remaining steps in processing the I/O request are performed by VMS I/O postprocessing. (See Section 4.3.1 for additional information.)

10.2 Timeout Handling Routines

VMS transfers control to the driver's timeout handling routine if a device unit does not request an interrupt within the time limit specified in the invocation of the wait-for-interrupt macro. Among its other activities, the VMS software timer interrupt service routine, having raised IPL from IPL$_TIMERFORK to IPL$_SYNCH, scans UCBs once every second to determine whether a device has timed out.

When the software timer interrupt service routine locates a device that has timed out, the routine calls the driver's timeout handling routine by performing the following steps:

1 It obtains both the fork lock and the device lock associated with the device unit to synchronize access to its fork database and device database. It raises IPL to device IPL as a result of obtaining the device lock.

2 It raises IPL on the local processor to IPL$_POWER to block local power failure servicing.

3 It disables expected interrupts and timeouts on the device by clearing bits in the status field of the device's UCB (UCB$V_INT and UCB$V_TIM in UCB$L_STS).

4 It sets the device-timeout bit in the UCB status field (UCB$V_TIMOUT in UCB$L_STS).

5 It lowers IPL to hardware device interrupt IPL (UCB$B_DIPL).

6 It restores the saved R3 and R4 of the driver's fork process from the UCB fork block (UCB$L_FR3 and UCB$L_FR4).

7 It restores R5 (address of the UCB fork block).

8 It computes the address of the driver's timeout handling routine from the saved PC in the UCB fork block (UCB$L_FPC).

9 It transfers control to the driver's timeout handling routine.

The driver's timeout handling routine executes in the following context:

- R0 through R5 are saved on the stack.
- R5 contains the address of the UCB for the device that timed out.
- Only system address space may be accessed.
- The processor is running in kernel mode.
- The processor is running on the interrupt stack.

- The processor holds both fork lock and device lock.
- IPL is at hardware device interrupt level.

A timeout handling routine returns control to the software timer interrupt service routine by issuing an RSB instruction. The driver's fork process eventually regains control, with R3 and R4 restored from UCB$L_FR3 and UCB$L_FR4.

Certain timeout handling routines may find it useful to fork to execute low priority code or to accomplish certain tasks, such as the restarting of an I/O request (see Section 10.2.1). If a driver uses this method, its interrupt service routine should check the interrupt-expected bit (UCB$V_INT) before handling the interrupt. The operating system clears this bit before it calls the driver's timeout handling routine. This allows the routine to determine whether device-timeout processing is in progress at fork IPL.

During recovery from a power failure, VMS forces a device timeout by altering the timeout field (UCB$L_DUETIM) of a UCB if that device's UCB records that the unit is waiting for an interrupt or timeout (UCB$V_INT and UCB$V_TIM set in UCB$L_STS). The timeout handling routine can perceive that recovery from a power failure is occurring by examining the power bit (UCB$V_POWER in UCB$L_STS) in the UCB.

A timeout handling routine usually performs one of three functions:

- It retries the I/O operation unless a retry count is exhausted.
- It aborts the I/O request, returning status (for instance, SS$_TIMEOUT) in R0.
- It sends a message to an operator mailbox and waits for a subsequent interrupt or timeout.

10.2.1 Retrying an I/O Operation

Some devices might retry an I/O operation after a timeout. For example, a disk driver's timeout handling routine might take the following steps after a transfer timeout:

1. Invokes the FORK macro to lower IPL to fork level.
2. Releases any owned map registers, data path, and controller data channel.
3. Determines whether it is possible to retry the I/O operation.
4. Examines the error retry count (UCB$B_ERTCNT) to determine whether it is possible to retry the I/O operation.

   If the retry count is exhausted, the timeout handling routine sets the error code, performs a normal abort I/O cleanup operation, and issues the REQCOM macro to complete the I/O request.

   If the retry count is not exhausted, the routine proceeds to the next step.

5 Examines the power bit (UCB$V_POWER in UCB$L_STS) to determine if it must take special steps before retrying the operation. For instance, the timeout handling routine should load the address of the IRP into R3 and reload the following fields of the IRP into the corresponding UCB fields, if they have been altered by partial processing of the I/O request:

    IRP$L_BCNT
    IRP$W_BOFF
    IRP$L_SVAPTE

    These actions set up an environment in which the transfer can be retried from the beginning.

6 Calls ERL$DEVICTMO to log the device timeout if the driver supports error logging (see Section 6.2).

7 Decreases the error retry count (UCB$B_ERTCNT).

8 Clears the UCB timeout bit (UCB$V_TIMOUT) in UCB$L_STS.

9 Branches to the start-I/O routine to retry the operation.

10.2.2 Aborting an I/O Request

A driver's timeout handling routine aborts the I/O request when it exhausts its retry count or when, having read device registers, the driver determines that some fatal error condition has occurred such that there is no point in retrying the request. Similarly, the routine aborts a request if the device's cancel-I/O bit (UCB$V_CANCEL in UCB$L_STS) is set, signifying that a cancel-I/O request was made.

To abort an I/O request, a timeout handling routine performs the following sequence of steps:

1 Clears the device control and status register (CSR), if appropriate to the device and controller

2 Invokes the FORK macro to lower IPL to fork level

3 Releases any owned map registers, data path, and controller data channel

4 Loads the abort status code (SS$_ABORT) into the low word of R0

5 Clears bits 16 through 31 in R0 to indicate that no data was transferred

6 Issues the REQCOM macro to complete the request

10.2.3 Sending a Message to the Operator

The following sequence describes a timeout handling routine that sends a message to the operator's mailbox and then goes back into a wait-for-interrupt or timeout state on the presumption that subsequent human intervention will make the device operational:

1 The timeout handling routine invokes the FORK macro to lower IPL to driver fork level.

2 It checks the cancel-I/O bit in the UCB status longword (UCB$V_CANCEL in UCB$L_STS).

If UCB$V_CANCEL is set, the timeout handling routine can abort the request. However, if UCB$V_CANCEL is clear, the timeout handling routine performs the following actions:

a. Saves R3 and R4 on the stack.

b. Loads an OPCOM message code, such as MSG$_DEVOFFLIN, into R4. Note that the driver must invoke the message definition macro $MSGDEF (located in SYS$LIBRARY:STARLET.MLB) to use these message codes.

c. Loads the address of the operator's mailbox (a pointer to which is located at SYS$AR_OPRMBX) into R3.

d. Calls a VMS routine to place the message in the operator's mailbox, as follows:

```
JSB    G^EXE$SNDEVMSG
```

e. Restores R3 and R4.

f. Invokes the DEVICELOCK macro to raise IPL to device IPL and obtain the associated device lock.

g. Issues a SETIPL macro to raise IPL$_POWER and prevent power failure interrupts on the local processor.

h. Invokes the WFIKPCH macro to wait for another interrupt or timeout.

When the OPCOM process reads the message in its mailbox, it sends the requested message, in this case "device-offline," to all operator terminals enabled for that device class.

11 Other Driver Routines

Drivers normally contain initialization, cancel-I/O, error logging, and register dumping routines. The driver prologue table specifies the addresses of the unit and controller initialization routines.[1] The driver dispatch table (DDT) contains the addresses of the cancel-I/O, error logging, and register dumping routines. The type of device determines which of these routines are required in a driver.

Drivers more rarely require a driver unloading routine, cloned UCB routine, or unit delivery routine. VMS, however, provides a method for specifying these routines in the DPT or DDT. A brief discussion of the driver unloading routine appears in Section 12.2.3. Section 11.4 describes the functions of a cloned UCB routine. A description of the unit delivery routine appears in Section 12.4.2.

11.1 Initialization Routines

Most device controllers and device units require initialization both when the corresponding device driver is loaded and when the operating system is recovering from a power failure. At these times, the duty of initialization routines is to prepare controllers and device units for operation, according to their characteristics.

The VMS operating system always calls controller and unit initialization routines with IPL raised to IPL$_POWER. The high IPL prevents any interrupts from reaching the local processor while initialization is occurring; for this reason, initialization routines should only contain code that is absolutely needed at initialization time. Initialization routines should *not* explicitly lower IPL. The system calls initialization routines with a JSB instruction; the routines return by executing an RSB instruction.

11.1.1 Controller Initialization Routine

The duties of a controller initialization routines depend on the characteristics of the device. For example, a controller initialization routine for a card reader might enable interrupts from the device by setting the interrupt-enable bit in the device's control and status register (CSR). A disk's controller initialization routine, on the other hand, might enable interrupts and initialize all unit-status registers. A controller initialization routine can typically perform any of the following tasks:

- Determine if it is being called during power failure recovery by examining the power bit (UCB$V_POWER in UCB$L_STS) in the

[1] A MASSBUS device driver must specify the address of its unit initialization routine in the driver dispatch table (using the **unitinit** argument to the DDTAB macro as discussed in Section 6.2). UNIBUS, Q22 bus, and generic VAXBI device drivers can specify the address in either the DPT or DDT.

UCB. A controller initialization routine may want to perform or avoid specific tasks during power failure recovery (see Section 11.1.4).

- Clear error-status bits in device registers.
- Initiate a device operation, such as clearing a drive or acknowledging a disk pack.
- Enable controller interrupts.
- If the controller is dedicated to a single-unit device, such as a printer, fill in IDB$L_OWNER and set the online bit (UCB$V_ONLINE in UCB$L_STS).
- Permanently allocate driver resources, such as
  - — UNIBUS/Q22 bus map registers (see Section 14.2.2.2)
  - — UNIBUS buffered data path (see Section 14.2.1.2)
- Allocate a buffer from nonpaged system dynamic memory.

Note that the permanent allocation of driver resources and the allocation of nonpaged pool require that the controller initialization routine fork to the driver's fork IPL. This action warrants careful coordination of the activities of the controller and unit initialization routines, both with each other and with the System Generation Utility (SYSGEN). See Section 11.1.5 for a discussion of forking in an initialization routine.

The controller initialization routine for a generic VAXBI device driver must initialize the device-specific aspects of the VAXBI device. Hardware initialization might include such activities as writing values to BIIC and device-specific registers, examining the results of the BIIC self-test, mapping a node's window space, building data structures to control the device, and linking these structures into chains of similar data structures. (Section 16.5 extensively discusses the means by which a driver's controller initialization routine performs these tasks.)

At the time of a call to a controller initialization routine, the following registers contain the listed values:

Register	Value
R4	Address of CSR
R5	Address of IDB that describes the controller
R6	Address of DDB associated with the controller
R8	Address of CRB for the controller

A controller initialization routine must preserve the contents of all registers except R0, R1, and R2.

11.1.2 Unit Initialization Routine

A unit initialization routine is useful for initializing device-dependent fields in the UCB. For example, a unit initialization routine for a disk can also specify disk-drive geometry (such as number of cylinders) in the UCB and wait for online units to spin up to speed. Unit initialization routines must set the online bit in the UCB (UCB$V_ONLINE) to declare the unit to be on line.

A unit initialization routine can perform the same types of tasks as a controller initialization routine (see Section 11.1.1). Generally, the driver for a single-unit controller does not need a unit initialization routine.

At the time of a call to a unit initialization routine, the registers contain the following values:

Register	Value
R3	Address of primary CSR
R4	Address of secondary CSR; R4 is equal to R3 if there is no secondary CSR
R5	Address of the device's UCB

A unit initialization routine must preserve the contents of all registers except R0, R1, and R2.

11.1.3 Initialization During Driver Loading

Prior to calling the initialization routines within a driver, VMS takes steps to initialize the appropriate I/O database structures and establish the appropriate links between these data structures and the driver. First, during system initialization, VMS creates an ADP for the device adapter. For generic VAXBI devices and MASSBUS devices, VMS creates an ADP, CRB, and IDB for the device at this time. Secondly, during driver loading, VMS performs some additional initialization. Finally, the driver's initialization routines are given an opportunity to initialize the device in a device-specific manner.

The extent of the initialization VMS performs during driver loading depends upon whether the I/O database is being created, and whether the driver is being loaded for the first time or is replacing a driver that was previously loaded.

The SYSGEN commands LOAD, AUTOCONFIGURE, and CONNECT add new drivers to the system configuration. The RELOAD command unloads an existing version of a driver and replaces it with a new one.

The LOAD command loads the driver into nonpaged system memory but does not call any driver-specific routines nor execute any initialization requests specified in DPT_STORE macro invocations nor create all of the I/O data structures associated with the device.

11.1 Initialization Routines

The AUTOCONFIGURE and CONNECT commands create and initialize I/O database structures associated with the device driver, call driver-specific initialization routines, and perform requests specified in DPT_STORE macro invocations. For each new device they add to the system, AUTOCONFIGURE and CONNECT perform the following steps:

- Create a UCB for the device. If this is the first occurrence of device and controller name, the commands create a DDB, CRB, and an IDB. (Because the CRB and IDB for a generic VAXBI device driver or MASSBUS device driver have already been created by the VMS adapter initialization routine, a CONNECT or AUTOCONFIGURE command for such a device never creates these structures.)
- Perform the initialization operations specified by the DPT_STORE macros within the initialization and reinitialization portions of the DPT.
- Relocate all addresses in the DDT and FDT to system virtual addresses.
- Call the controller initialization routine specified in the CRB, if it has created a CRB (or if CRB$V_UNINIT is set in CRB$B_MASK for a generic VAXBI device).
- Call the unit initialization routine (if any) specified in the DDT. If no routine exists in the DDT, call the unit initialization routine (if any) specified in the CRB.

The AUTOCONFIGURE and CONNECT command operations raise IPL to IPL$_POWER before calling the driver's initialization routines.

The RELOAD command replaces an existing driver with a new driver. The command loads the new driver's code into nonpaged system memory. Unlike the other SYSGEN commands for driver loading, RELOAD assumes that the data structures associated with the driver already exist, and thus updates the I/O database to reflect the modified code and its different location in system virtual address space. It performs the following functions:

- Calls the driver unloading routine in the old version of the driver, if one exists (as indicated in the **unload** argument of the DPTAB macro) and if bit DPT$V_NOUNLOAD in DPT$B_FLAGS is clear.

  The driver unloading routine must return success status in R0 for SYSGEN to proceed with the following steps.
- Deallocates the memory occupied by the old version of the driver.
- Loads the new version of the driver.
- Executes requests specified by DPT_STORE macro invocations in only the reinitialization section of the DPT in the new driver.
- Relocates all addresses in the FDT and DDT to system virtual addresses.
- Calls the controller initialization routine.

Chapter 12 contains detailed descriptions of all SYSGEN commands related to device drivers.

11.1.4 Initialization During Recovery from a Power Failure

During recovery from a power failure, the operating system locates every UCB in the I/O database, by following the chain of pointers to all DDBs in the system (starting at IOC$GL_DEVLIST and chained by DDB$L_LINK) and the chain of pointers to all UCBs of the same device and controller type (starting at DDB$L_UCB and chained by UCB$L_LINK). For each UCB it finds, VMS performs the following procedure:

1 It locates the CRB associated with the UCB (UCB$L_CRB) and determines whether a controller initialization routine exists for the device's controller by examining CRB$L_INTD+VEC$L_INITIAL. If an invocation of the DPT_STORE macro loaded the address of a controller initialization routine into this field, VMS calls that routine.

2 It determines whether a unit initialization routine exists for the particular device unit by examining the unit initialization field of the DDT (DDT$L_UNITINIT). If the field does not contain an address, the system checks the CRB (CRB$L_INTD+VEC$L_UNITINIT).[2]

  If either the CRB or the DDT contains a nonzero address for such a routine, the system calls the routine to initialize the device unit. The system calls only one routine; if the DDT contains an address, the address in the CRB is ignored.

When called during power failure recovery, driver initialization routines must adhere to the following rules:

- These routines cannot acquire any spin locks. Controller and unit initialization routines are called at IPL 31 during power failure recovery to reinitialize I/O devices before the processors are allowed to proceed with execution at lower IPLs. Because processors may have been holding spin locks at the time of the power failure, they will not be able to release them until after they resume execution. As a result, spin locks are not available to controller and unit initialization routines.
- They cannot perform any operation that requires the intervention of other processors in a VMS multiprocessing system.

A driver initialization routine can determine if it is being called during power failure recovery by examining the power bit (UCB$V_POWER in UCB$L_STS) in the UCB.

[2] MASSBUS drivers store unit initialization routines addresses only in the DDT.

11.1.5 Forking from a Driver Initialization Routine

If a driver initialization routine must fork to perform a thread of code that must synchronize with code or a structure synchronized at a lower IPL, it must take special care to avoid breaking that synchronization.

First of all, because SYSGEN, under normal circumstances, immediately calls a driver's unit initialization routine at IPL$_POWER after its controller initialization completes, the unit initialization routine must be prepared for the instance of a controller initialization routine that forks. Such a unit initialization routine would complete before the fork thread of the controller initialization routine resumed.

A fork thread in a unit initialization routine (or a controller initialization routine in a driver without a unit initialization routine) must otherwise take the following precautions to avoid breaking synchronization:

- Use either the CRB fork block, or a fork block defined in a device-specific extension to the UCB. The separate fork block prevents a conflict with the use of the normal UCB fork block by the IOFORK routine. If you are using a separate UCB fork block, you must not attempt to allocate the fork block from paged pool.
- You should use a semaphore bit to protect against multiple forking. Remember that the unit initialization routine may be called repeatedly if multiple power failures require servicing. If the semaphore shows that a fork is in progress, then exit without attempting to fork. Access the semaphore bit using interlocked instructions (for example, BBSSI or BBCCI).
- Invoke EXE$FORK with R5 pointing to the alternate fork block. Restore the original value of R5 once the fork process is active.
- Restore all registers on exit. Because EXE$FORK removes the caller's address from the stack and returns to the caller's caller, the unit initialization routine must set up a dummy caller's caller routine to restore registers destroyed by EXE$FORK.

11.2 Cancel-I/O Routine

VMS routines call a device driver's cancel-I/O routine under the following circumstances:

- When a process issues a Cancel-I/O-on-Channel system service ($CANCEL)
- When a process deallocates a device, causing the device reference count (UCB$W_REFC) to become zero (that is, no process I/O channels are assigned to the device)

- When a process deassigns a channel from a device, using the $DASSGN system service[3]
- When VMS performs cleanup operations as part of image termination by canceling all pending I/O requests for the image and closing all image-related files open on process I/O channels

The VMS routine EXE$CANCEL locates the UCB for the device associated with a process I/O channel from a pointer in the CCB, as follows:

channel index number → CCB → UCB

EXE$CANCEL performs the following steps:

1 Obtains the fork lock associated with the driver, thus raising IPL to fork IPL.

2 Removes from the device's pending-I/O queue all IRPs associated with the process and that channel.

3 For a buffered-I/O read operation, clears the buffered-read function bit (IRP$V_FUNC) in IRP$W_STS.

4 Sets the status code SS$_CANCEL in IRP$L_MEDIA.

5 Inserts the IRPs removed from the pending-I/O queue into the systemwide I/O postprocessing queue.

6 Requests a software interrupt from the local processor at IPL$_IOPOST.

7 Calls the cancel-I/O routine specified in the DDT of the associated device driver (argument **cancel** to the DDTAB macro). EXE$CANCEL locates the routine using the following chain of pointers:

UCB → DDT → cancel-I/O routine

The cancel-I/O routine gives the driver an opportunity to prevent further device-specific processing of the I/O request currently being processed on the device.

11.2.1 Context of a Cancel-I/O Routine

When EXE$CANCEL calls the cancel-I/O routine, the local processor is at driver fork IPL holding the associated fork lock. As a result, the cancel-I/O routine can read and modify the device's UCB. Registers at the time of the call contain the following values:

[3] Note that if the call to $DASSGN deassigns the last channel to the device, the device driver's cancel-I/O routine is called a second time. Channel deassignment and last channel deassignment are both potentially significant events for certain devices. The former means, in effect, that a user has finished with a device and the latter means that all users are finished with a device. The reason code for both events is CAN$C_DASSGN. However, a driver's cancel-I/O routine can distinguish between the two cases by examining UCB$W_REFC.

Register	Value	
R2	Channel index number.	
R3	Contents of UCB$L_IRP (address of current IRP, if any, for device).	
R4	Address of process control block (PCB) of process for which the $CANCEL system service is being performed.	
R5	Address of device's UCB.	
R8	Reason for call to cancel the I/O request. Codes that signify the reasons for cancellation are defined by the $CANDEF macro. Possible values for R8 include	
	CAN$C_CANCEL	Called by $CANCEL system service
	CAN$C_DASSGN	Called by $DASSGN or $DALLOC system service

If a cancel-I/O routine uses registers other than R0 through R3, it must save the registers and restore them before exiting.

Device drivers might want to base their cancel-I/O operation on whether the cancel-I/O request is the result of a channel deassignment (CAN$C_DASSGN). For example, the terminal driver cancels out-of-band AST requests only if the call to its cancel-I/O routine results from a Deassign-I/O-Channel ($DASSGN) system service call.

11.2.2 Drivers That Need No Cancel-I/O Routine

Some devices do not need any device-dependent processing performed for an I/O request; you can omit the **cancel** argument from the DDTAB macro. In this case, the DDTAB macro expansion loads the address of the VMS routine IOC$RETURN into the appropriate position in the DDT. The routine IOC$RETURN executes a single RSB instruction.

11.2.3 Device-Independent Cancel-I/O Routine

Drivers can specify the VMS routine IOC$CANCELIO as the value of the **cancel** argument in the DDTAB macro invocation. IOC$CANCELIO cancels I/O to a device in the following device-independent manner:

1 It confirms that the device is busy by examining the device-busy bit in the UCB status longword (UCB$V_BSY in UCB$L_STS).

2 It locates the process-identification field in the IRP currently being processed on the device by using the following chain of pointers:

UCB → IRP → process identification field

IOC$CANCELIO confirms that the field (IRP$L_PID) contains the same value as the corresponding field in the PCB (PCB$L_PID).

3 It confirms that the specified channel-index number is the same as the value stored in the IRP's channel-index field (IRP$W_CHAN).

4 It sets the cancel-I/O bit in the UCB status longword (UCB$V_CANCEL in UCB$L_STS). Other driver routines, such as the timeout handling routine, check the cancel-I/O bit to determine whether to retry the I/O operation or abort it. (See Section 10.2.2 for additional information.)

11.2.4 Device-Dependent Cancel-I/O Routine

Drivers that include their own cancel-I/O routines must perform the first three steps of IOC$CANCELIO listed in Section 11.2.3 to determine whether the I/O request being processed originates from the process canceling I/O on a channel. If the three checks succeed, the cancel-I/O routine can proceed in a device-specific manner. For instance, a cancel-I/O routine may perform the following tasks:

- Clear UCB$V_INT and UCB$V_TIM in the UCB status longword (UCB$L_STS)
- Release any owned map registers, data path, and controller data channel
- Load a status code (SS$_CANCEL, for instance) into the low word of R0
- Load other status information into the high word of R0 and the longword of R1
- Issue the REQCOM macro to complete the request

11.3 Error Logging Routines

A driver that supports error logging must satisfy the following prerequisites:

- It must invoke the data structure definition macro $EMBDEF (located in SYS$LIBRARY:LIB.MLB).
- It must use the local disk extension or local tape extension of the UCB. These extensions include error-log extension. (See the data structures in the *VMS Device Support Reference Manual* for additional information.)
- It must provide a means whereby error logging can be enabled for the device. For instance, it can use the DPT_STORE macro to set the device characteristic DEV$V_ELG in UCB$L_DEVCHAR or it can support an IO$_SETCHAR function that sets this bit.
- It must ensure that the size of the error log buffer, as specified in DDT$W_ERRORBUF, is large enough to accommodate EMB$L_DV_REGSAV+4, plus one longword for each register to be dumped. It must specify this value in the **erlgbf** argument to the DDTAB macro.
- It should include a register dumping routine, specifying its address in the **regdmp** argument of the DDTAB macro.

- It must complete the servicing of the I/O request by invoking the REQCOM macro. (Routines, like ERL$DEVICEATTN, that log errors that are not associated with the current I/O request skip this step.) IOC$REQCOM takes steps to complete the error logging initiated by a call to an error logging routine.

11.3.1 Error Logging Routines Supplied by VMS

The VMS operating system provides the following routines that drivers can call to allocate and fill an error message buffer after a device error or timeout occurs:

Routine	Function
ERL$DEVICERR	Logs an error associated with the I/O request in progress
ERL$DEVICTMO	Logs a timeout associated with the I/O request in progress
ERL$DEVICEATTN	Logs an error not associated with an I/O request

These routines are described in full in the *VMS Device Support Reference Manual*, but they all perform similar functions, as follows:

- Increment UCB$W_ERRCNT to record a device error. If the error-log-in-progress bit (UCB$V_ERLOGIP in UCB$L_STS) is set, the routine returns control to its caller (ERL$DEVICERR and ERL$DEVICTMO only).
- Allocate from the current error log allocation buffer an error message buffer of the length specified in the device's DDT (in argument **erlgbf** to the DDTAB macro).
- Initialize the buffer with the current system time, error log sequence number, and error type code. These routines use the following error type codes:

Routine	Error Code
ERL$DEVICERR	Device error (EMB$C_DE)
ERL$DEVICTMO	Device timeout (EMB$C_DT)
ERL$DEVICEATTN	Device attention (EMB$C_DA)

- Place the address of the error message buffer in UCB$L_EMB.
- Set UCB$V_ERLOGIP in UCB$L_STS.
- Load into R0 the address of the location in the buffer in which the contents of the device registers are to be stored.
- Call the driver's register dumping routine.

11.3.2 Register Dumping Routine

A driver that supports error logging or diagnostics specifies the address of a register dumping routine in the **regdmp** argument to the DDTAB macro.

When an error logging routine passes control to the driver's register dumping routine, the following registers contain the listed values:

Register	Value
R0	Address of buffer into which a register dumping routine copies the contents of device registers
R4	Address of device's CSR (if the driver invoked the WFIKPCH macro to wait for an interrupt or timeout)
R5	Address of UCB

The register dumping routine preserves the contents of all registers except R0 through R2. If it uses the stack, the register dumping routine must restore the stack before passing control to another routine, waiting for an interrupt, or returning control to its caller.

A register dumping routine uses the following procedure to fill the indicated buffer:

1 Writes a longword value representing the number of device registers to be written into the buffer.

2 Moves device register longword values into the buffer following the register count longword.

The source of these register values depends upon the nature of the driver. If the driver has established a UCB extension, its interrupt service routine can copy to it the values of critical device registers. In this case, the register dumping routine may contain instructions similar to the following:

```
MOVL    UCB$L_TD_STATUS(R5),(R0)+
```

Alternatively, the register dumping routine can obtain device register values directly from I/O address space, offsetting from the address of the CSR as follows:

```
MOVZWL TD_STATUS(R4),(R0)+
```

Note that this latter method is not truly accurate in that, at the time a register dumping routine runs, all or some device registers may have been modified during the servicing of device interrupts unrelated to the error.

When a driver fork process invokes the system routine IOC$DIAGBUFILL, as described in the *VMS Device Support Reference Manual*, the routine transfers control to the register dumping routine with the address of the diagnostic buffer in R0, the address of the device's CSR in R4, and the address of the UCB in R5.

11.3.3 Interpreting Error Log Entries

See the *Guide to Maintaining a VMS System* and the *VMS Error Log Utility Manual* for help with producing and reading error log files.

11.4 Cloned UCB Routine

EXE$ASSIGN calls the driver's cloned UCB routine when an Assign I/O Channel system service request ($ASSIGN) specifies a template device (that is, bit UCB$V_TEMPLATE in UCB$L_STS is set). EXE$ASSIGN does not assign the channel to the template device itself. Rather, it creates a copy of the template device's UCB and ORB, initializing and clearing certain fields as appropriate.

A cloned UCB routine receives control at IPL$_ASTDEL in kernel mode with process context available, holding the I/O database mutex (IOC$GL_MUTEX).

Only drivers for network devices or template devices, such as mailboxes, include a cloned UCB routine. A driver specifies the address of a cloned UCB routine in the **cloneducb** argument of the DDTAB macro.

The driver's cloned UCB routine verifies the contents of fields in the UCB and ORB and completes their initialization. When a cloned UCB routine is called, the following locations contain the listed values:

Location	Contents
R0	SS$_NORMAL
R2	Address of cloned UCB
R3	Address of DDT
R4	Address of current PCB
R5	Address of template UCB
UCB$L_FQFL(R2)	Address of UCB$L_FQFL(R2)
UCB$L_FQBL(R2)	Address of UCB$L_FQBL(R2)
UCB$L_FPC(R2)	0
UCB$L_FR3(R2)	0
UCB$L_FR4(R2)	0
UCB$W_BUFQUO(R2)	0
UCB$L_ORB(R2)	Address of cloned ORB
UCB$L_LINK(R2)	Address of next UCB in DDB chain
UCB$L_IOQFL(R2)	Address of UCB$L_IOQFL(R2)
UCB$L_IOQBL(R2)	Address of UCB$L_IOQBL(R2)
UCB$W_UNIT(R2)	Device unit number (minimum UCB$W_UNIT_SEED(R5)+1)
UCB$W_CHARGE(R2)	Mailbox byte quota charge (UCB$W_SIZE)
UCB$W_REFC(R2)	0
UCB$L_STS(R2)	UCB$V_DELETEUCB set, UCB$V_ONLINE set

Location	Contents
UCB$W_DEVSTS(R2)	UCB$V_DELMBX set if DEV$V_MBX is set in UCB$L_DEVCHAR(R2)
UCB$L_OPCNT(R2)	0
UCB$L_SVAPTE(R2)	0
UCB$W_BOFF(R2)	0
UCB$W_BCNT(R2)	0
UCB$L_ORB(R2)	Address of cloned ORB
ORB$L_OWNER of template ORB	UIC of current process
ORB$L_ACL_MUTEX of template ORB	$FFFF_{16}$
ORB$B_FLAGS of template ORB	ORB$V_PROT_16 set
ORB$W_PROT of template ORB	0
ORB$L_ACL_COUNT of template ORB	0
ORB$L_ACL_DESC of template ORB	0
ORB$R_MIN_CLASS of template ORB	0 in first longword

A cloned UCB routine must preserve the contents of R2 and R4. It issues an RSB instruction to return control to EXE$ASSIGN. If the routine returns error status in R0, EXE$ASSIGN undoes the process of UCB cloning and completes with the failure status in R0.

12 Loading a Device Driver

You can load a non-Digital-supplied device driver any time after the system is bootstrapped. If the driver contains an error and the error does not crash or corrupt the operating system, you can correct the error and reload a new version of the driver.

12.1 Preparing a Driver for Loading into the Operating System

To prepare a device driver for loading, perform the following steps:

1. Write the device driver in one or more source files. If the driver comprises several source files, you must insert a .PSECT directive before any generated code in all files except the file that contains the DPTAB and DDTAB macro invocations. The following .PSECT must be used:

   ```
   .PSECT   $$$115_DRIVER
   ```

   If a single source file contains the driver, you must not specify any .PSECT directives. The declaration of the DPTAB and DDTAB macros correctly establishes driver program sections ($$$105_PROLOGUE and $$$115_DRIVER, respectively).

2. Assemble the source files with the system's macro library (SYS$LIBRARY:LIB.MLB). For example:

   ```
   $  MACRO MYDRIVER.MAR+SYS$LIBRARY:LIB.MLB/LIBRARY
   ```

3. Link the object file with the VMS global symbol table, which is located in SYS$SYSTEM and called SYS.STB. If the driver consists of several source files, you must specify the file that contains the driver prologue table as the first file in the list. The linker-options file must contain a BASE statement specifying a zero base for the executable image. The following is an example of the creation of the options file and the LINK command used to link a driver:

   ```
   $  CREATE MYDRIVER.OPT
   BASE=0
   Ctrl/Z
   $  LINK /NOTRACE MYDRIVER1[,MYDRIVER2,...],-
   _$         MYDRIVER.OPT/OPTIONS,-
   _$         SYS$SYSTEM:SYS.STB/SELECTIVE_SEARCH
   ```

   The resulting image must consist of a single image section. The linker will report that the image has no transfer address.

Once you have linked or relinked a driver, you should copy its image to the SYS$LOADABLE_IMAGES or SYS$SYSTEM directory. The SYSGEN LOAD and CONNECT commands first search for a driver in the SYS$LOADABLE_IMAGES directory. If they do not find the driver, they then search the SYS$SYSTEM directory.

12.2 Loading a Driver

Once the driver has been linked correctly, it is ready to be loaded. To load the driver into system virtual memory, run the System Generation Utility (SYSGEN) from the system manager's account or from an account having CMKRNL privilege, using the following command:

```
$ RUN SYS$SYSTEM:SYSGEN
```

SYSGEN responds with a prompt and waits for further input:

```
SYSGEN>
```

The *VMS System Generation Utility Manual* describes the full set of SYSGEN commands. The sections that follow describe those commands SYSGEN uses to load drivers:

SYSGEN Command	Privilege Required
LOAD	CMKRNL
CONNECT	CMKRNL
RELOAD	CMKRNL
SHOW/ADAPTER	CMEXEC
SHOW/BI	CMEXEC
SHOW/BUS	CMEXEC
SHOW/CONFIGURATION	CMEXEC
SHOW/DEVICE	CMEXEC
SHOW/XMI	CMEXEC

SYSGEN takes special steps to ensure that drivers that do not adhere to multiprocessing synchronization standards do not coexist in a system with drivers that are properly synchronized. The procedure that SYSGEN follows to accomplish this is discussed in Section 12.3. In addition, SYSGEN provides an automatic configuration service for most all devices of the various bus-types, as described in Section 12.4.

12.2.1 LOAD Command

To load a device driver, issue the LOAD command.

Note: If the controller has only a single unit attached to it, you can issue the CONNECT command to perform the driver-loading tasks normally performed by the LOAD command, as well as its task of creating the device's I/O database (see Section 12.2.2).

Format

LOAD filespec

Parameter

filespec
Name of a file containing an executable driver image. The driver-loading procedure compares the file specification from the command line with the names of the drivers in the current system configuration. If the procedure discovers that a driver with the same name already exists in the configuration, it will not load the new driver. If it does not find a configured driver with the same name, it loads the new driver into contiguous locations in nonpaged pool, and links the DPT into the system's linked list of DPTs (headed by IOC$GL_DPTLIST).

The LOAD command uses SYS$LOADABLE_IMAGES as the default device/directory name, and EXE as the default file type. If it cannot find the driver in the SYS$LOADABLE_IMAGES directory, it searches for it in SYS$SYSTEM.

Example

```
SYSGEN>  LOAD CRDRIVER
```

This command loads the driver found in SYS$LOADABLE_IMAGES:CRDRIVER.EXE (the card-reader driver).

12.2.2 CONNECT Command

The CONNECT command creates data structures in the I/O database for a specified device. The device-connecting procedure performs the following general functions:

- If the CONNECT command specifies a new device unit on an *existing* controller, it creates a unit control block for the new unit and calls the driver's unit initialization routine.
- If the CONNECT command specifies a device unit on a *new* controller, it creates a device data block, channel request block, interrupt dispatch block, and unit control block and then calls both the controller initialization and unit initialization routine in the driver. (Note that, because system initialization creates the CRB and IDB for a generic VAXBI device, the CONNECT command for such a device omits the creation and initialization of these structures.)

The CONNECT command can also load into system memory a driver that has not been previously loaded. (See the following discussion of the /DRIVERNAME qualifier and the description of the LOAD command in Section 12.2.1 for information on driver loading.)

Caution: The database-loading procedure does little error checking. If you specify a vector that has already been defined, the procedure rejects the CONNECT command. However, if the CONNECT command specifies an incorrect CSR address, the I/O database is apt to become corrupted and will likely cause a system failure.

Format

CONNECT device

Parameter

device

Name of the device to be connected. Specify the device name in the format *ddcu* where

dd = device code (up to 9 alphabetic characters)

c = controller designation (alphabetic)

u = unit number

For example, LPA0 specifies the line printer (LP) on controller A at unit number 0. When specifying the device name, do *not* follow it with a colon (:).

The device code and controller specification must be a unique and accurate device name and controller combination. If data structures for the specified device/controller already exist, the device-connecting procedure does not create any data structures or perform any initialization operations. If the device/controller name does not accurately name a device, the procedure creates spurious data structures.

The device-connecting procedure examines the I/O database for data structures that support the specified device. The procedure creates the following data structures if they do not exist:

- DDB for the specified device/controller combination (*ddcu*).
- CRB and IDB for the specified controller. The device-connecting procedure creates these data structures whenever it creates a DDB for a UNIBUS, MASSBUS, or Q22 bus device.
- UCB for the device unit. The device-connecting procedure creates a UCB whenever it creates a DDB, or when a UCB for the specified device does not exist. If a UCB already exists, the procedure ceases its modifications to the I/O database and continues its other tasks.

After it creates these data structures, the procedure initializes them as follows:

- Performs the initialization operations specified by the DPT_STORE macros in the initialization and reinitialization portions of the DPT.
- Relocates all addresses in the DDT and FDT to absolute system virtual addresses.
- Raises IPL to IPL$_POWER on the local processor so that initialization is not interrupted.
- If it created a new CRB (or is connecting a generic VAXBI device), calls the controller initialization routine, if one is specified by CRB$L_INTD+VEC$L_INITIAL.
- Calls the unit initialization routine if one is specified by DDT$L_UNITINIT. If the DDT$L_UNITINIT does not specify a unit initialization routine, the device-connecting procedure calls the unit initialization routine (if any) specified by CRB$L_INTD+VEC$L_UNITINIT.

Required Qualifiers

/[NO]ADAPTER=nexus
Nexus value of the UNIBUS adapter, MASSBUS adapter, or other controller to which the device unit is attached. The nexus can be a number or a generic name as listed by the /ADAPTER qualifier to the SYSGEN command SHOW. (See Section 12.2.4 for a discussion of the SHOW/ADAPTER command.) For generic VAXBI devices, this value is the VAXBI node number (see Section 12.2.5 for discussion of the SHOW/BI command).

Table 12–1 lists typical nexus assignments for UNIBUS and MASSBUS adapters. For XMI bus and nexus assignments, refer to Sections 12.2.6 and 12.2.7 for a discussion of the SHOW/BUS and SHOW/XMI commands.

Table 12–1 Conventional Nexus Assignments

Adapter	VAX–11/730	VAX–11/750	VAX–11/780 VAX–11/785 VAX 8600/8650	VAX 82*x*0/83*x*0 VAX 6000 series VAX 85*x*0/8700/88*x*0
UNIBUS				
0	3	8	3	0
1	–	9	4	16
2	–	–	5	–
3	–	–	6	–
MASSBUS				
0	–	4	8	–
1	–	5	9	–
2	–	6	10	–
3	–	–	11	–

All numeric values are interpreted as decimal unless they are preceded by a radix descriptor (%O or %X). Issue the CONNECT command with the /NOADAPTER qualifier to connect drivers associated with software devices. The mailbox driver is an example of this type of driver.

/CSR=csr-addr
UNIBUS or Q22 bus address of the device's control and status register (CSR). All numeric values are interpreted as decimal unless they are preceded by a radix descriptor (%O or %X). Table 12–2 provides additional information on vector and CSR assignments for UNIBUS and Q22 bus devices.

/CSR_OFFSET=value
Offset from the CSR address of a multiple-device controller board to the CSR address of the device. The /CSR_OFFSET qualifier is only required for a multidevice board, such as the DMF32. All numeric values are interpreted as decimal unless they are preceded by a radix descriptor

(%O or %X). Table 12–2 provides additional information on vector and CSR assignments for UNIBUS and Q22 bus devices.

/VECTOR=vector-addr

Q22 bus or UNIBUS address of the interrupt vector for the device. All numeric values are interpreted as decimal unless they are preceded by a radix descriptor (%O or %X). Table 12–2 provides additional information on vector and CSR assignments for UNIBUS and Q22 bus devices.

/VECTOR_OFFSET=value

Offset from the interrupt vector of a multiple-device board to the interrupt vector of the device being connected. The /VECTOR_OFFSET qualifier is only required for a multi-device board, such as the DMF32. All numeric values are interpreted as decimal unless they are preceded by a radix descriptor (%O or %X). Table 12–2 provides additional information on vector and CSR assignments for UNIBUS and Q22 bus devices.

Optional Qualifiers

/NUMVEC=vector-cnt

Number of interrupt vectors for the device. If this qualifier is omitted, the default number of vectors is 1. The number specified by the /VECTOR qualifier is the address of the lowest vector. Vectors must be contiguous.

/DRIVERNAME=driver

Name of the driver for the device to be connected. If the driver for the specified device has not yet been loaded, the CONNECT command will load its driver. First, it will attempt to load the driver whose name is specified in this qualifier, defaulting to a file type of EXE in device/directory SYS$LOADABLE_IMAGES. (If it cannot find the driver in SYS$LOADABLE_IMAGES, the CONNECT command checks SYS$SYSTEM.)

If the /DRIVERNAME qualifier is omitted, CONNECT follows one of two procedures to supply a default name. If the device to be connected is the first unit on the controller, CONNECT concatenates the first two characters of the device code with "DRIVER" (for example, LPDRIVER). Otherwise, CONNECT obtains the driver name from the field DDB$T_DRVNAME.

Consult the SYSGEN device table in Table 12–2 for the driver names of the devices supported by VMS.

/ADPUNIT=unit-number

Unit number of a device on the MASSBUS adapter. The unit number for a disk drive is the number of the plug on the drive. For magnetic tape drives, the unit number corresponds to the tape controller's number.

/MAXUNITS=max-unit-cnt

Maximum number of units attached to the controller. This number determines the size of the UCB list appended to the IDB. If specified, this value overrides the maximum number of units designated in the DPT. The maximum number of units is stored in the field IDB$W_UNITS.

Example

```
SYSGEN>  CONNECT LPA0 /ADAPTER=UB0/CSR=%O777514/VECTOR=%O200
```

This command loads the driver LPDRIVER, if it is not already loaded, and creates the data structures (DDB, CRB, IDB, and UCB) needed to describe LPA0. It also causes the driver's controller and unit initialization routines to be executed.

12.2.3 RELOAD Command

The RELOAD command loads a driver and removes a previously loaded version of that driver.

The RELOAD command provides all of the functions of LOAD, except that it loads the driver regardless of whether it is already loaded. If any of the units associated with the driver is busy, the driver cannot be reloaded; SYSGEN issues an error message.

Caution: Use the RELOAD command only when all devices supported by the driver are inactive. The checks for activity made by the RELOAD command might not detect all device activity, and changing a driver while an I/O request is being processed will cause a system failure.

Format

RELOAD filespec

Parameter

filespec
Name of a file containing an executable driver image. The driver-reloading procedure compares the name DPT$T_NAME of the driver being loaded with the names of the drivers in the current system configuration. If no such driver is configured, the driver-reloading procedure loads the driver as described in the discussion of the LOAD command in Section 12.2.1.

If the SYSGEN reloading procedure finds a driver with the specified name in the configuration, it first determines that the current driver can be replaced in the following steps:

- Confirms that the DPT$V_NOUNLOAD flag of the current driver is not set.
- Ensures that no devices that use the current driver are busy, as indicated by the UCB$V_BSY bit set in UCB$L_STS.
- If these checks succeed, calls the current driver's driver unloading routine, if one has been specified in the **unload** argument of the DPTAB macro.

  The driver unloading routine executes in process context at IPL$_POWER. It cannot lower IPL or obtain spin locks.

  At the time of the call, register R10 contains the address of the DPT.

A driver unloading routine can take steps to ensure that no thread of code or structure exists in the system that may reference the space occupied by the version of the driver about to be unloaded, a timer-queue element (TQE), for instance.

A driver unloading routine can use COM$DRVDEALMEM to return system buffers allocated by the driver to nonpaged pool. The driver unloading routine returns status in R0 to the driver-reloading procedure.

Upon receiving success status, the driver-reloading procedure replaces the current driver with the new driver in the following manner:

1 Loads the new driver into contiguous locations in nonpaged pool.

2 Searches the I/O database for references to the driver. If any DDB refers to the driver being reloaded, the driver-reloading procedure must reinitialize data structure fields according to the reinitialization instructions in the new DPT (see Section 6.1).

Fields that must be reinitialized when a driver is reloaded include those that contain relative addresses within the driver:

- Addresses of the interrupt service routines
- Addresses of the unit and controller initialization routines
- Address of the driver dispatch table

3 Calls the driver's controller initialization routine. (It does not call the unit initialization routine.)

4 Removes the newly replaced driver from the system's linked list of DPTs (headed by IOC$GL_DPTLIST) and deallocates the nonpaged system space the old driver occupied.

5 Links the address of the new DPT to the system's DPT list.

12.2.4 SHOW/ADAPTER Command

The SHOW/ADAPTER command displays nexus numbers and generic names of UNIBUS adapters, VAXBI adapters, memory controllers, and interconnection devices such as the DEBNI and CI. Use of the SHOW /ADAPTER command requires CMEXEC privilege.

Format

SHOW/ADAPTER

Example

```
SYSGEN>  SHOW/ADAPTER

CPU Type:  VAX 6000-360
```

Nexus	(decimal)	Generic Name or Description
0001	1	XMI - 6000-200/300 processor
0002	2	XMI - 6000-200/300 processor
0003	3	XMI - 6000-200/300 processor
0004	4	XMI - 6000-200/300 processor
0005	5	XMI - 6000-200/300 processor
0006	6	XMI - 6000-200/300 processor
0007	7	XMI - memory module
0008	8	XMI - memory module
0009	9	XMI - memory module
000A	10	XMI - memory module
000B	11	XMI - memory module
000C	12	XMI - memory module
		XMI - BI Adapter (DWMBA/A)
		XMI - BI Adapter (DWMBA/A)
00D1	209	BI - XMI Adapter (DWMBA/B)
00D3	211	CIO
00D6	214	BI - NI Adapter (DEBNI)
00E1	225	BI - XMI Adapter (DWMBA/B)
00E4	228	BI - Disk Adapter (KDB50)
00E6	230	BI - TK50 Adapter (TBK50)

12.2.5 SHOW/BI Command

The SHOW/BI command displays device addresses that are currently mapped in the I/O space for the VAXBI bus. It also displays node and nexus numbers and generic names of UNIBUS adapters, VAXBI adapters, memory controllers, and interconnection devices such as the DMB32 and CI. Use of the SHOW/BI command requires CMEXEC privilege.

Format

SHOW/BI

Example 1

```
SYSGEN> SHOW/BI

CPU Type:  VAX 8800                          Cpu Connection: NMI

     ** Bus map for BI 00 on 28-AUG-1990 14:13:02.95 **
 Address 20000000 (node 00) responds with value 0108 CI
 Address 20004000 (node 02) responds with value 0106 BI - NMI Adapter (NBIB)
 Address 2000E000 (node 07) responds with value 0109 BI Combo Board (DMB32)
     ** Bus map for BI 01 on 28-AUG-1990 14:13:03.00 **
 Address 22000000 (node 00) responds with value 0102 UB
 Address 22004000 (node 02) responds with value 0106 BI - NMI Adapter (NBIB)
 Address 2200E000 (node 07) responds with value 410F BI - NI Adapter (DEBNA)
```

Example 2

```
SYSGEN> SHOW/BI

CPU Type:  VAX 6000-450                        Cpu Connection: XMI

     ** Bus map for BI 00 on 28-AUG-1990 14:14:54.03 **
 Address 3A002000 (node 01) responds with value 2107 BI - XMI Adapter (DWMBA/B)
```

12.2.6 SHOW/BUS Command

The SHOW/BUS command displays the hierarchical bus structure of a system and all attached devices. The display list includes node numbers, generic names of processors, memory modules, adapters such as VAXBI adapters, memory controllers, and interconnection devices such as the NI. Use of the SHOW/BUS command requires CMEXEC privilege.

Format

SHOW/BUS

Example 1

```
SYSGEN>  SHOW/BUS

  Cpu Type: VAX 8800                          Cpu Connection: NMI
  Bus      Node    Generic Name              Nexus(hex)  Connection Address

 BI 00     00     CI                           0000
 BI 00     02     BI - NMI Adapter (NBIB)      0002
 BI 00     07     BI Combo Board (DMB32)       0007

 BI 01     00     UB                           0010
 BI 01     02     BI - NMI Adapter (NBIB)      0012
 BI 01     07     BI - NI Adapter (DEBNA)      0017
```

Example 2

```
SYSGEN>  SHOW/BUS

  Cpu Type: VAX 6000-450                      Cpu Connection: XMI
  Bus      Node    Generic Name              Nexus(hex)  Connection Address

 XMI 00    01     XMI - 6000-400 processor     0001
 XMI 00    02     XMI - 6000-400 processor     0002
 XMI 00    03     XMI - 6000-400 processor     0003
 XMI 00    04     XMI - 6000-400 processor     0004
 XMI 00    05     XMI - 6000-400 processor     0005
 XMI 00    06     XMI - memory module          0006
 XMI 00    07     XMI - memory module          0007
 XMI 00    08     XMI - memory module          0008
 XMI 00    09     XMI - memory module          0009
 XMI 00    0A     XMI - memory module          000A
 XMI 00    0B     XMI - memory module          000B
 XMI 00    0C     XMI - NI adapter (DEMNA)     000C
 XMI 00    0D     XMI - BI Adapter (DWMBA/A)   000D Connects to BI 00 node 01

   BI 00    01     BI - XMI Adapter (DWMBA/B)    00D1 Connects to XMI 00 node 0D

 XMI 00    0E     XMI - BI Adapter (DWMBA/A)   000E Connects to BI 01 node 01

   BI 01    01     BI - XMI Adapter (DWMBA/B)    00E1 Connects to XMI 00 node 0E
   BI 01    02     BI - Disk Adapter (KDB50)     00E2
   BI 01    04     CI                            00E4
   BI 01    06     BI - TK50 Adapter (TBK50)     00E6
```

12.2.7 SHOW/XMI Command

The SHOW/XMI command displays device addresses that are currently mapped in the I/O space for the XMI bus. It also displays node and nexus numbers and generic names of processors, adapters, VAXBI adapters, memory controllers, and interconnection devices such as the NI. Use of the SHOW/XMI command requires CMEXEC privilege.

Format

SHOW/XMI

Example

```
SYSGEN>  SHOW/XMI

     ** Bus map for XMI 00 on 28-AUG-1990 14:14:50.48 **
 Address 21880000 (node 01) responds with value 8082 XMI - 6000-400 processor
 Address 21900000 (node 02) responds with value 8082 XMI - 6000-400 processor
 Address 21980000 (node 03) responds with value 8082 XMI - 6000-400 processor
 Address 21A00000 (node 04) responds with value 8082 XMI - 6000-400 processor
 Address 21A80000 (node 05) responds with value 8082 XMI - 6000-400 processor
 Address 21B00000 (node 06) responds with value 4001 XMI - memory module
 Address 21B80000 (node 07) responds with value 4001 XMI - memory module
 Address 21C00000 (node 08) responds with value 4001 XMI - memory module
 Address 21C80000 (node 09) responds with value 4001 XMI - memory module
 Address 21D00000 (node 0A) responds with value 4001 XMI - memory module
 Address 21D80000 (node 0B) responds with value 4001 XMI - memory module
 Address 21E00000 (node 0C) responds with value 0C03 XMI - NI adapter (DEMNA)
 Address 21E80000 (node 0D) responds with value 2001 XMI - BI Adapter (DWMBA/A)
 Address 21F00000 (node 0E) responds with value 2001 XMI - BI Adapter (DWMBA/A)
```

12.2.8 SHOW/CONFIGURATION Command

The SHOW/CONFIGURATION command displays the device name, number of units, nexus number and type, and shows the CSR and vector addresses of devices connected to or autoconfigured in the system.

Format

SHOW/CONFIGURATION

Optional Qualifiers

/ADAPTER=nexus

Nexus value of the UNIBUS adapter, MASSBUS adapter, or other interconnect to be displayed. The nexus value can be expressed as an integer or as one of the generic names listed by the SHOW/ADAPTER command.

/COMMAND_FILE

Option by which you instruct SYSGEN to format all device data produced by the SHOW/CONFIGURATION command into CONNECT /ADAPTER=nexus commands and write them to a specified output file. By executing the commands in this file, you can remove a device from floating address space without completely reconnecting the CSR and vector addresses of the remaining devices. See the *VMS System Generation Utility Manual* for more details.

/OUTPUT=filespec
Name of a file into which SHOW/CONFIGURATION is to write device configuration information.

Example

```
SYSGEN>  SHOW/CONFIGURATION/ADAPTER=UB1

     System CSR and Vectors on  24-JUL-1991 14:58:26.08

Name: LPA  Units: 1 Nexus:4 (UBA) CSR: 777514 Vector1: 200 Vector2: 000
Name: DYA  Units: 2 Nexus:4 (UBA) CSR: 777170 Vector1: 264 Vector2: 000
Name: XMA  Units: 1 Nexus:4 (UBA) CSR: 760070 Vector1: 300 Vector2: 304
Name: XMB  Units: 1 Nexus:4 (UBA) CSR: 760100 Vector1: 310 Vector2: 314
Name: XMC  Units: 1 Nexus:4 (UBA) CSR: 760110 Vector1: 320 Vector2: 324
Name: TTA  Units: 8 Nexus:4 (UBA) CSR: 760130 Vector1: 330 Vector2: 334
Name: TTB  Units: 8 Nexus:4 (UBA) CSR: 760140 Vector1: 340 Vector2: 344
Name: TTC  Units: 8 Nexus:4 (UBA) CSR: 760150 Vector1: 350 Vector2: 354
Name: TTD  Units: 8 Nexus:4 (UBA) CSR: 760160 Vector1: 360 Vector2: 364
Name: TTE  Units: 8 Nexus:4 (UBA) CSR: 760170 Vector1: 370 Vector2: 374
```

12.2.9 SHOW/DEVICE Command

The SHOW/DEVICE command displays the following information:

- Name of the driver
- Starting virtual address of the driver (that is, the address of its DPT)
- Ending virtual address of the driver
- Generic device/controller name associated with the driver
- Addresses of the DDB, CRB, and IDB for the generic device/controller supported by the driver
- Unit number and UCB address of each device unit associated with the driver

The SHOW/DEVICE command requires CMEXEC privilege.

Format

SHOW/DEVICE [=driver-name]

Parameter

driver-name
Name of the driver for which the information is to be displayed. If a driver name is not specified, the command displays information about all drivers and devices known to the system.

Example

```
SYSGEN>  SHOW/DEVICE=TMDRIVER

__DRIVER___START____END___DEV___DDB______CRB______IDB_______UNIT___UCB

TMDRIVER 8009DF00 8009F020
                          MTA 800BA660 800BA6C0 800BA360
                                                        0 8009F020
                                                        1 8009F0C0
```

12.3 Loading Uniprocessing and Multiprocessing Drivers

In a VMS multiprocessing environment, the presence of a device driver that does not adhere to multiprocessing synchronization conventions might impair proper system functions. VMS takes steps to either prohibit the enabling of multiprocessing in a VAX system that has such a driver present or prevent the loading of such a driver if multiprocessing has already been enabled.

To accomplish this, the VMS driver-loading routine assumes that any driver that can run in a VMS multiprocessing environment uses the spin lock synchronization macros and loads the appropriate I/O database fields. (See Section E.3 for information on how to produce a driver that can execute in a VMS multiprocessing environment.) Use of the spin lock synchronization macros causes VMS to set the SMP-modified bit in the DPT (DPT$V_SMPMOD in DPT$L_FLAGS).

If multiprocessing has *not* been enabled on the system, the driver-loading routine checks the SMP-modified bit in the DPT and takes either of the following actions:

- If the SMP-modified bit is set, the driver-loading routine loads the driver and calls its controller and unit initialization routines, as discussed in Section 12.2.
- If the SMP-modified bit is *not* set, the driver-loading mechanism sets the unmodified-driver bit (SMP$V_UNMOD_DRIVER) in SMP$GL_FLAGS, thus prohibiting the subsequent enabling of multiprocessing on the system. It then loads the driver and calls its controller and unit initialization routines. If such a driver has been successfully loaded into a VMS system, you cannot subsequently enable multiprocessing.

If multiprocessing is currently enabled on the system, the driver-loading mechanism checks the SMP-modified bit in the DPT and takes either of the following actions:

- If the SMP-modified bit is set, the driver-loading mechanism loads the driver and calls its controller and unit initialization routines.
- If the SMP-modified bit is *not* set, the driver-loading mechanism does not load the driver, returning the error status SS$_NONSMPDRV to its caller.

12.4 The SYSGEN Autoconfiguration Facility

Traditionally, SYSGEN is invoked near the end of system initialization processing during the execution of the system startup command procedure. This procedure generally issues a SYSGEN AUTOCONFIGURE ALL command, the result of which is that SYSGEN scans various device tables to determine devices VMS expects to be connected to each UNIBUS, Q22 bus, MASSBUS, and VAXBI bus configured in the system. Ultimately, as the autoconfigure facility discovers the data structures associated with the devices recognized by VMS, it loads the associated device drivers and invokes their initialization routines.

To configure devices attached to the UNIBUS or Q22 bus, SYSGEN goes through the steps described in subsequent sections of this chapter. Because the autoconfigure facility cannot recognize non-Digital-supplied VAXBI devices, the system startup procedure (or a later invocation of SYSGEN) must explicitly request that SYSGEN connect the device.[1] SYSGEN responds to such explicit requests by utilizing the data structures created by the VMS adapter initialization module for the unknown VAXBI device to load the associated device driver and invoke its initialization routines.

SYSGEN automatically configures a UNIBUS or Q22 bus as follows:

- It initializes the base of floating space to 300_8 and 760010_8 for vectors and CSRs, respectively.
- It tests fixed and floating CSR address space for all known Digital devices.
- When a device is found at a CSR, SYSGEN reserves floating CSR and vector space for that device, if necessary.
- It searches for the name of the driver associated with the device by checking the SYSGEN device table (shown in Table 12–2) and the directory SYS$LOADABLE_IMAGES (or SYS$SYSTEM). If the driver has already been loaded or exists as an image file in SYS$LOADABLE_IMAGES (or SYS$SYSTEM), SYSGEN creates and initializes the I/O database for that device and loads the driver's image if necessary. If the device at the CSR is supported by VMS and SYSGEN cannot locate its associated driver's image, it generates an error message. If the device is unsupported and has no corresponding driver's image, SYSGEN ignores the condition.

12.4.1 SYSGEN Device Table

Digital-supplied devices are attached to the UNIBUS or Q22 bus according to the following basic rules:

- A device of type A is always at a fixed and predefined CSR address; the device always interrupts at a fixed and predefined vector address; only one example of device A can be configured in each system.
- A device of type B is identical to type A except that 1 through *n* examples can be configured in a single system. Examples 2 through *n* are also located at fixed and predefined CSRs and vector addresses.
- Devices of type C (1 through *n* of them) are always at fixed and predefined CSR addresses; however, the interrupt vector addresses vary according to what other devices are present on the system.
- Devices of type D (1 through *n* of them) are at CSR addresses and vector addresses that vary according to what other devices are present on the system.

[1] Because the autoconfigure facility will never be called for a non-Digital-supplied device, any unit delivery routine that a VAXBI device driver may include will never be called.

CSR and vector addresses that vary are called floating addresses. The devices must be located in floating CSR and vector space according to the order in which the devices appear in the SYSGEN device table. This table, shown in Table 12–2, lists all the type A and type B devices supported by VMS. It also lists the type C and type D devices that are recognized by SYSGEN's autoconfiguration procedure.

For UNIBUS and Q22 bus systems, the base of floating CSR space is 760010_8. CSR range 764100_8 to 767776_8 is reserved for Digital's Customer Special Systems (CSS) and for non-Digital-supplied devices. Reserved addresses 766000_8 to 767776_8 are recommended for customer third-party devices. For UNIBUS and Q22 bus systems, the base of floating vector space is 300_8. For third-party devices, it is recommended that the vector assignment begin from the top of floating vector space and work down toward its base.

Table 12–2 lists the characteristics of all devices recognized by SYSGEN. This table indicates the following information for each device type:

- Device name
- Device controller name
- Interrupt vector
- Number of interrupt vectors per controller
- Vector alignment factor
- Address of the first device register for each controller recognized by SYSGEN. (The first register is usually, but not always, the CSR.)
- Number of registers per controller
- Device driver name
- Indication of whether the driver is supported

Devices not listed in the SYSGEN device table include the following:

- Non-Digital-supplied devices with fixed CSR and vector addresses. These devices have no effect on autoconfiguration. Customer-built devices should be assigned CSR and vector addresses beyond the floating address space reserved for Digital-supplied devices.
- Those Digital-supplied floating-vector devices that the AUTOCONFIGURE command does not recognize. Use the CONNECT command to attach these devices to the system.

Table 12–2 SYSGEN Device Table

Device Name	Controller Name	Vector	Number of Vectors	Vector Alignment	CSR/Rank	Register Alignment	Driver Name	Support
CR	CR11	230	1	—	777160	—	CRDRIVER	Yes
DM	RK611	210	1	—	777440	—	DMDRIVER	Yes
LP	LP11	200 170 174 270 274	—	—	777514 764004 764014 764024 764034	—	LPDRIVER	Yes
DL	RL11	160	1	—	774400	—	DLDRIVER	Yes
MS	TS11	224	1	—	772520	—	TSDRIVER	Yes
DY	RX211	264	1	—	777170	—	DYDRIVER	Yes
DQ	RB730	250	1	—	775606	—	DQDRIVER	Yes
PU	UDA	154	1	—	772150	—	PUDRIVER	Yes
PT	TU81	260	1	—	774500	—	PUDRIVER	Yes
XE	UNA	120	1	—	774510	—	XEDRIVER	Yes
XQ	QNA	120	1	—	774440	—	XQDRIVER	Yes
OM	DC11	Float	2	8	774000 774010 774020 774030 . . . 32 units maximum	—	OMDRIVER	No
DD	TU58	Float	2	8	776500 776510 776520 776530 . . . 16 units maximum	—	DDRIVER	Yes
OB	DN11	Float	1	4	775200 775210 775220 775230 . . . 16 units maximum	—	OBDRIVER	No

(continued on next page)

Table 12–2 (Cont.) SYSGEN Device Table

Device Name	Controller Name	Vector	Number of Vectors	Vector Alignment	CSR/Rank	Register Alignment	Driver Name	Support
YM	DM11B	Float	1	4	770500 770510 770520 770530 . . . 16 units maximum	—	YMDRIVER	No
OA	DR11C	Float	2	8	767600 767570 767560 767550 . . . 16 units maximum	—	OADRIVER	No
PR	PR611	Float	1	8	772600 772604 772610 772614 . . . 8 units maximum	—	PRDRIVER	No
PP	PP611	Float	1	8	772700 772704 772710 772714 . . . 8 units maximum	—	PPDRIVER	No
OC	DT11	Float	2	8	777420 777422 777424 777426 . . . 8 units maximum	—	OCDRIVER	No

(continued on next page)

Table 12–2 (Cont.) SYSGEN Device Table

Device Name	Controller Name	Vector	Number of Vectors	Vector Alignment	CSR/Rank	Register Alignment	Driver Name	Support
OD	DX11	Float	2	8	776200 776240	—	ODDRIVER	No
YL	DL11C	Float	2	8	775610 775620 775630 775640 . . . 31 units maximum	—	YLDRIVER	No
YJ	DJ11	Float	2	8	Float	8	YJDRIVER	No
YH	DH11	Float	2	8	Float	16	YHDRIVER	No
OE	GT40	Float	4	8	772000 772010	—	OEDRIVER	No
LS	LPS11	Float	6	8	770400	—	LSDRIVER	No
OR	DQ11	Float	2	8	Float	8	ORDRIVER	No
OF	KW11W	Float	2	8	772400	—	OFDRIVER	No
XU	DU11	Float	2	8	Float	8	XUDRIVER	No
XV	DV11	Float	3	8	775000 775040 775100 775140	—	XVDRIVER	No
OG	LK11	Float	2	8	Float	8	OGDRIVER	No
XM	DMC11	Float	2	8	Float	8	XMDRIVER	Yes
TTA	DZ11	Float	2	8	Float	8	DZDRIVER	Yes
XK	KMC11	Float	2	8	Float	8	XKDRIVER	No
OH	LPP11	Float	2	8	Float	8	OHDRIVER	No
OI	VMV21	Float	2	8	Float	8	OIDRIVER	No
OJ	VMV31	Float	2	8	Float	16	OJDRIVER	No
OK	DWR70	Float	2	8	Float	8	OKDRIVER	No
DL	RL11	Float	1	4	Float	8	DLDRIVER	Yes
MS	TS11	Float	1	4	772524 772530 772534	—	TSDRIVER	Yes
LA	LPA11	Float	2	8	770460	—	LADRIVER	Yes
LA	LPA11	Float	2	8	Float	16	LADRIVER	Yes
OL	KW11C	Float	2	8	Float	8	OLDRIVER	No
DY	RX211	Float	1	4	Float	8	DYDRIVER	Yes

(continued on next page)

Table 12–2 (Cont.) SYSGEN Device Table

Device Name	Controller Name	Vector	Number of Vectors	Vector Alignment	CSR/Rank	Register Alignment	Driver Name	Support
XA	DR11W	Float	1	4	Float	8	XADRIVER	Yes
XB	DR11B	124	—	—	772410	—	XBDRIVER	No
XB	DR11B	Float	1	4	772430	—	XBDRIVER	No
XB	DR11B	Float	1	4	Float	8	XBDRIVER	No
XD	DMP11	Float	2	8	Float	8	XDDRIVER	Yes
ON	DPV11	Float	2	8	Float	8	ONDRIVER	No
IS	ISB11	Float	2	8	Float	8	ISDRIVER	No
XD	DMV11	Float	2	8	Float	16	XDDRIVER	No
XE	UNA	Float	1	4	Float	8	XEDRIVER	No
XQ	QNA	Float	1	4	774460	—	XQDRIVER	Yes
PU	UDA	Float	1	4	Float	4	PUDRIVER	Yes
XS	KMS11	Float	3	8	Float	16	XSDRIVER	No
XP	PCL11	Float	2	8	764200 764240 764300 764340	—	XPDRIVER	No
VB	VS100	Float	1	4	Float	16	VBDRIVER	No
PT	TU81	Float	1	4	Float	4	PUDRIVER	Yes
OQ	KMV11	Float	2	8	Float	16	OQDRIVER	No
UK	KCT32	Float	2	8	764400 764440 764500 764540	—	UKDRIVER	No
IX	IEQ11	Float	2	8	764100	—	IXDRIVER	No
TX	DHV11	Float	2	8	Float	16	YFDRIVER	Yes
DT	TC11	214	1	—	777340	—	DTDRIVER	No
VC	VCB01	Float	2	1	777200	—	VCDRIVER	Yes
VC	VCB01	Float	2	1	Float	64	VCDRIVER	Yes
OT	LNV11	Float	1	4	776200	—	OTDRIVER	No
LD	LNV21	Float	1	4	Float	16	LDDRIVER	No
ZQ	QTA	Float	1	4	772570	—	ZQDRIVER	No
ZQ	QTA	Float	1	4	Float	8	ZQDRIVER	No
SJ	DSV11	Float	1	4	Float	8	SJDRIVER	No
OU	ADV11C	Float	2	8	Float	8	OUDRIVER	No
OV	AAV11	Float	0	8	770440	—	OVDRIVER	No
OV	AAV11C	Float	0	8	Float	8	OVDRIVER	No

(continued on next page)

Table 12–2 (Cont.) SYSGEN Device Table

Device Name	Controller Name	Vector	Number of Vectors	Vector Alignment	CSR/Rank	Register Alignment	Driver Name	Support
AX	AXV11C	140	2	—	776400	—	AXDRIVER	No
AX	AXV11C	Float	2	8	Float	8	AXDRIVER	No
KZ	KWV11C	Float	2	8	770420	—	KZDRIVER	No
KZ	KWV11C	Float	2	8	Float	4	KZDRIVER	No
AZ	ADV11D	Float	2	8	776410	—	AZDRIVER	No
AZ	ADV11D	Float	2	8	Float	4	AZDRIVER	No
AY	AAV11D	Float	2	8	776420	—	AYDRIVER	No
AY	AAV11D	Float	2	8	Float	4	AYDRIVER	No
VA	VCB02	Float	3	16	777400 777402 777404 777406 . . . 8 units maximum	—	VADRIVER	Yes
DN	DRV11J	Float	16	4	764160 764140 764120	—	DNDRIVER	No
HX	DRQ3B	Float	2	8	Float	16	HXDRIVER	No
VQ	VSV24	Float	1	4	Float	8	VQDRIVER	No
VV	VSV21	Float	1	4	Float	8	VVDRIVER	No
BQ	IBQ01	Float	1	4	Float	8	BQDRIVER	No
UT	MIRA	Float	2	8	Float	8	UTDRIVER	No
IX	IEQ11	Float	2	8	Float	16	IXDRIVER	No
AW	ADQ32	Float	2	8	Float	32	AWDRIVER	No
VX	DTC04	Float	2	8	Float	2	VXDRIVER	No
CQ	DESNA	Float	1	4	Float	32	CQDRIVER	No
GQ	IGQ11	Float	2	8	Float	4	GQDRIVER	No
RS	KMV1F	Float	2	8	Float	32	RSDRIVER	No
SD	DIV32	Float	1	8	Float	4	SDDRIVER	No
VN	DTCN5	Float	2	8	Float	4	VNDRIVER	No
VM	DTCO5	Float	2	8	Float	4	VMDRIVER	No
UJ	KWV32	Float	2	8	Float	8	UJDRIVER	No
PK	QZA	Float	1	4	Float	64	PKIDRIVER	Yes

12.4.2 Device Driver Control of Autoconfiguration

The SYSGEN autoconfiguration facility provides two features that drivers can use to control the automatic configuration of the devices they operate. These features are invoked through the **defunits** and **deliver** arguments to the DPTAB macro.

The **defunits** argument to the DPTAB macro specifies a default number of units to be configured on each controller. The DPTAB macro copies this value to the DPT$W_DEFUNITS field in the DPT. The SYSGEN autoconfiguration facility reads this field and creates UCBs numbered zero through the default unit number minus one. The default value of **defunits** is 1.

The **deliver** argument to the DPTAB macro specifies the address of a driver-specific unit delivery routine. An offset to this routine is stored in the DPT$W_DELIVER field in the DPT. When the **deliver** argument is present, the SYSGEN autoconfiguration facility calls the unit delivery routine once for each unit, the number of which is specified in the **defunits** argument.

The unit delivery routine prevents the creation of UCBs for devices that do not respond to a test for their presence.

If the unit delivery routine returns a true status in R0, the unit is configured. If the status in R0 is false, the autoconfiguration facility does not configure the device. If the **deliver** argument is not used, the unit delivery feature is disabled.

SYSGEN calls the unit delivery routine with a JSB instruction in the following context:

- Interrupt priority level is at IPL$_POWER.
- R0 through R2 are available for use.
- R3 contains the address of the IDB, if one exists. If none exists, the value contained in R3 is zero.
- R4 contains the address of the CSR for the controller.
- R5 contains the number of the unit that the routine must decide whether or not to configure.
- R6 contains the base address of UNIBUS adapter I/O space.
- R7 contains the address of the configuration control block (ACF).
- R8 contains the address of the ADP.

The configuration control block (ACF) is described in the *VMS Device Support Reference Manual*.

A driver may or may not specify a unit delivery routine. For instance, the DZ11's device driver specifies 8 as the default unit number, but provides no routine to configure eight terminal units automatically for each DZ11's CSR. The RK611 device driver specifies 8 as the default number of units and also specifies the address of a unit delivery routine that is called once for each of the eight possible devices on the controller.

12.4.3 Floating-Vector Address Calculation

To calculate the floating-vector address of a device, SYSGEN rounds the current floating-vector base (CFVB) up to the next valid vector address boundary for the next device in the table.

If a device is present, SYSGEN reserves floating-vector space for the device by computing a new CFVB:

CFVB + (4 * number-of-vectors) → CFVB

12.4.4 Floating-CSR Address Calculation

To calculate the floating CSR address of a device, SYSGEN rounds the current floating CSR base (CFCB) up to the next valid floating CSR address. Floating CSR addresses must be on a word boundary.

SYSGEN tests the CSR address (CFCB) for the next device in the device table by executing a TSTW instruction on the address and noting whether there is a response at that address.

If the device is present, SYSGEN reserves floating CSR address space for the device by computing a new CFCB:

CFCB + bytes-in-register-set → CFCB

When all devices of a particular type have been located and their floating CSR space reserved, SYSGEN reserves an extra block of CSR space to indicate a change to a new device type:

CFCB + 2 → CFCB

If the device is not present, SYSGEN reserves an extra block of CSR space to indicate a change to a new device type by adding two to the rounded CFCB:

CFCB + 2 → CFCB

12.4.5 Rules for Configuration

The formulas described in Sections 12.4.3 and 12.4.4 reduce to the following rules:

- Devices with fixed CSR addresses and fixed vector addresses must be attached according to the SYSGEN device table settings.
- Devices with floating CSR or vector addresses must be attached in the order in which they are listed in the SYSGEN device table.
- A 2-byte gap must be reserved between each device type located in floating CSR address space.
- A 2-byte gap must be reserved in floating CSR address space for each device type that has no controller in its configuration.
- An extra 2-byte gap must be reserved between the KW11C and the RX11 in floating CSR address space.

When assigning floating vector addresses and registers to devices not supplied by Digital, be sure to leave a generous gap between these addresses and those of devices in the table because future VMS maintenance updates might add new devices to the SYSGEN device table.[2]

[2] UNIBUS addresses 766000_8 through 767776_8 are available for non-Digital-supplied devices.

13 Debugging a Device Driver

DELTA and XDELTA are debugging tools that can be used to monitor the execution of user programs and the VMS operating system. When you link DELTA with a user image that runs in a nonprivileged process, DELTA is a user-mode debugging tool. When run in a privileged process, however, DELTA acts as a multimode debugger; it allows you to debug in user mode or to change to kernel mode for debugging. However, DELTA does not support debugging at elevated IPLs.

XDELTA is syntactically identical to DELTA but also allows you to debug code that executes at an elevated IPL. XDELTA is used for standalone debugging of driver code and the executive.

This chapter primarily describes the use of XDELTA as a tool for debugging an executing driver image.

The chapter includes discussions of two additional topics:

- Detection and analysis of driver errors in a VMS multiprocessing system
- Detection of corruption in nonpaged pool and the ways in which the corrupting code can be discovered

These topics supplement information presented in the *VMS System Dump Analyzer Utility Manual.*

13.1 Bootstrapping the System with XDELTA

Under VMS, drivers are part of the operating system. You normally bootstrap the system with two boot flags set to allow you to debug with XDELTA. One flag causes the bootstrapping procedure to include XDELTA in the system. The other boot flag indicates a stop at the breakpoint at the beginning of VMS initialization. (The BREAKPOINTS system parameter, by default, enables a breakpoint at the end of system initialization. See Section 13.2 for additional information.) Table 13–1 describes the possible values of these flags. Following a boot that includes XDELTA, executing a BPT instruction causes control to transfer to a fault handler located in XDELTA.

13.1 Bootstrapping the System with XDELTA

Table 13–1 Boot Flags That Control the Loading of XDELTA

Flag Value (f)	Meaning
0	Normal nonstop bootstrap (default)
1	Stop in SYSBOOT (equivalent to @D*xy*GEN on the VAX–11/780)
2	Include XDELTA with the system but do not take the initial breakpoint
6	Include XDELTA with the system and take the initial breakpoint
7	Include XDELTA with the system, stop in SYSBOOT and take the initial breakpoint at system initialization (equivalent to @D*xy*XDT on the VAX–11/780)

The procedures for bootstrapping the system with XDELTA differ depending upon the system on which the operating system is running. Some VAX systems that use a console block storage device supply a special boot command file that automatically includes XDELTA in the system and causes the processor to stop in SYSBOOT and take the initial breakpoint at system initialization. When booting other systems, you must specify the appropriate flag value in the BOOT command. Table 13–2 lists some recommended methods for booting with XDELTA. See your system's installation and operations guides for additional information.

Table 13–2 Recommended Methods for Bootstrapping with XDELTA

Boot Commands	Explanation
MicroVAX/VAXstation Systems and VAX–11/750[1] Systems	

B[*/f*] *devname*	B is the console BOOT command. The flags (*f*) parameter is a 32-bit hexadecimal integer loaded into R5 as input to VMB.EXE, the primary bootstrap program. See Table 13–1 for a list of its possible values.
	Using the format *ddcu*, specify the name of the device that contains the volume to be bootstrapped. You must supply both controller (*c*) and unit (*u*) identifiers; there are no defaults. If you omit *devname*, the *f* parameter is ignored.
	The following example bootstraps a MicroVAX II system from DUA0.[2]
	`>>> B/7 DUA0`
	`SYSBOOT>`
	`SYSBOOT> CONTINUE`

[1]The console TU58 of the VAX–11/750 system contains command files (DMAXDT.CMD and DBAXDT.CMD) analogous to those supplied for the VAX–11/780. See your system's installation and operations guides for additional information.

[2]At the SYSBOOT prompt enter other required SYSBOOT commands and conclude the boot operation with a CONTINUE command. If you do not set or load system parameters with a USE command, the system uses parameters stored in the system image. To prevent the system from automatically rebooting after a bugcheck, clear the BUGREBOOT system parameter.

(continued on next page)

Table 13–2 (Cont.) Recommended Methods for Bootstrapping with XDELTA

Boot Commands	Explanation
VAX–11/730 Systems	

@DQAXDT
@DQ0XDT
	Use either DQAXDT.CMD or DQ0XDT.CMD, depending upon the boot device. The following example boots from DQA1, first depositing the value 1 in R3. When the boot device is DQA0, you can omit this step and execute DQ0XDT.COM.[2]
	
	`>>> D/G/L 3 1`
	`>>> @DQAXDT`
	`SYSBOOT>`
	`SYSBOOT> CONTINUE`

VAX 6000-Series Systems	

B /R5:*f* [*bootspec*]	B is the console BOOT command. The flags (*f*) parameter is a 32-bit hexadecimal integer loaded into R5 as input to VMB.EXE, the primary bootstrap program. See the XDELTA boot flags in Table 13–1 for a list of its possible values.
	
	The *bootspec* may indicate a specific boot path using explicit /XMI and /BI qualifiers, or it may be a symbolic name for qualifiers which you previously specified for your VAX 6000-series configuration. Refer to your system-specific operations guide for more information on booting and the boot specifications.
	
	If you do not currently use R5 in any boot specification, use the values in the XDELTA boot flag table for R5. For example, to boot from the boot disk at VAXBI node 4 through the DWMBA adapter at XMI node E, load XDELTA, stop in SYSBOOT, and take the initial breakpoint (R5=7) as follows:
	
	`>>> B/R5:7 /XMI:E /BI:4 DU1`
	`SYSBOOT>`
	
	If you have previously specified a /R5 qualifier value for your VAX 6000-series configuration, you must logically OR this value with one from the XDELTA boot table. For example, if you have a boot alias SYS1 defined to boot from system root SYS1, your boot specification might result as follows:
	
	`>>> SHOW BOOT`
	`SYS1  /XMI:E/BI:4/R5:10000000`
	
	For the XDELTA boot, you must then OR the values from the XDELTA boot flag table with your existing R5 specification. If you take the initial break point, in this example, specify the following:
	
	`>>> B SYS1 /R5:10000006`

[2]At the SYSBOOT prompt enter other required SYSBOOT commands and conclude the boot operation with a CONTINUE command. If you do not set or load system parameters with a USE command, the system uses parameters stored in the system image. To prevent the system from automatically rebooting after a bugcheck, clear the BUGREBOOT system parameter.

(continued on next page)

Table 13–2 (Cont.) Recommended Methods for Bootstrapping with XDELTA

Boot Commands	Explanation
VAX 8600/8650 Systems	

@DU0XDT	Use DU0XDT.COM, if available on the console media, according to the method described for the VAX–11/780. Otherwise, perform a normal bootstrap using the available *ddu*GEN.COM or *ddu*BOO.COM according to the following method:
	Use the */NOSTART* qualifier in the BOOT command to cause the processor to pause and await console commands after it boots. After a variety of progress messages are displayed, the console prompt reappears. First, determine a value for the flag that controls XDELTA loading (see Table 13–1). Then, examine the current value of R5; if it is nonzero (for instance, it is the system root number), perform an inclusive-OR operation upon it and your selected XDELTA flag value.[2]
	`>>> BOOT/NOSTART`
	`>>> EXAMINE R5`
	`>>> DEPOSIT R5 7`
	`>>> CONTINUE`
	`SYSBOOT>`
	`SYSBOOT> CONTINUE`

VAX 9000-Series Systems	

B[/R5:*f*]
[/XMI:*xmi_info*]
[/BI:*vaxbi_node_id*]
devname
	B is the console BOOT command. The flags (*f*) parameter is a 32-bit hexadecimal integer loaded into R5 as input to VMB9AQ.EXE, the primary bootstrap program. See the XDELTA boot flags in Table 13–1 for a list of its possible values.
	Specify *xmi_info* using the XMI number and XMI node number (both in hexadecimal) of the node being accessed. The XMI bus number defaults to zero. The hexadecimal number that you specify with this qualifier must be in the format *xy*, where *x* is the XMI bus number and *y* is the XMI node number.
	Specify *devname* in the format *ddduuu* and the VAXBI node ID of the device in the /BI qualifier. (You can substitute a symbolic name for these qualifiers as discussed in your system-specific operations guide.)
	The console passes the specified unit number (*uuu*) to the procedure *ddd*BOO.CMD. If you do not specify *devname*, the console executes DEFBOO.CMD.
	The following example bootstraps a VAX 9000 system from the boot disk (unit 0) at node 3 of the VAXBI bus at node 11 (C_{16}) of the XMI0 bus.[2]
	`>>> B/R5:7/XMI:C/BI:3 KDM0`
	`SYSBOOT>`
	`SYSBOOT> CONTINUE`

[2]At the SYSBOOT prompt enter other required SYSBOOT commands and conclude the boot operation with a CONTINUE command. If you do not set or load system parameters with a USE command, the system uses parameters stored in the system image. To prevent the system from automatically rebooting after a bugcheck, clear the BUGREBOOT system parameter.

13.2 Proceeding from the Initial Breakpoints

Before stopping at any breakpoints that may be defined in driver code, the VAX processor can stop at either or both of two breakpoints in system initialization.

The breakpoint at the end of system initialization is enabled by the default setting of the BREAKPOINTS system parameter. The breakpoint at the beginning is enabled by the appropriate value of the boot flag as described in Table 13–1.

After being bootstrapped, the system displays its welcoming message and halts in XDELTA, as follows:

```
1 BRK AT nnnnnnnn
address/NOP
```

XDELTA is waiting for input. (XDELTA never issues explicit prompts.) Usually, you proceed from this point with the following command:

```
;P  Return
```

All of the XDELTA commands are described in Section 13.10 and in the *VMS Delta/XDelta Utility Manual*.

If the operating system halts with a fatal bugcheck, the system prints the bugcheck information on the console terminal. Then, because the BUGREBOOT system parameter is clear, XDELTA prompts. Bugcheck information consists of the following:

- Type of bugcheck
- Register values
- Dump of one or more stacks

PC and stack content indicate how an experimental driver crashed the system. You can then examine the system state further by issuing XDELTA commands.

13.3 Loading the Driver

Once the system is running, you can log in to the system as the system manager and load the experimental driver.

To load the driver, run SYSGEN and issue the appropriate LOAD and CONNECT commands. Example 13–1 provides a sample dialogue.

The first SHOW command in Example 13–1 causes SYSGEN to display the location of the device driver in system memory. You then define the device to the operating system. The second SHOW command causes SYSGEN to display the driver's location and the addresses of the device's DDB, CRB, IDB, and UCB.

Example 13–1 Loading a Driver

```
$ RUN SYS$SYSTEM:SYSGEN
SYSGEN> LOAD DMA0:[YOUR.DIRECTORY]YRDRIVER.EXE
SYSGEN> SHOW /DEVICE=YRDRIVER

 Driver    Start    End    Dev   DDB      CRB      IDB     Unit  UCB
YRDRIVER   80060E50 80061070
SYSGEN> CONNECT YR /ADAP=3/VEC=%O274/CSR=%O776240
SYSGEN> SHOW /DEVICE=YRDRIVER

 Driver    Start    End    Dev   DDB      CRB      IDB     Unit  UCB
YRDRIVER   80060E50 80061070
                            YRA 8005FDC0 80060B70 8005FE00
                                                        0 80060BB0

SYSGEN> EXIT
```

13.4 Inserting Breakpoints in Driver Source Code

The SYSGEN command CONNECT calls controller initialization and unit initialization routines. To begin debugging the driver, you should ensure that the kernel-mode debugging utility XDELTA gains control of the driver before these routines execute. This is accomplished by placing one or more calls to the special system routine INI$BRK within the source code of either the controller or unit initialization routine. To call INI$BRK, use the following instruction:

```
JSB     G^INI$BRK
```

The INI$BRK routine contains two instructions:

```
BPT
RSB
```

When the processor executes the BPT instruction, XDELTA gains control and reports the address of the breakpoint:

```
1 BRK AT nnnnnnnn
```

You can use INI$BRK as a debugging tool and place calls to it within any part of the driver source code.

To determine the last driver PC before the breakpoint, examine the kernel stack. The stack register is register RE (hexadecimal format):

```
RE/ address  / address
```

Display RE to find the address of the top of the stack. Another display command (/) reveals the contents of the top of the stack, which should be the return address to the driver that called INI$BRK.

13.5 Calculating the Base of Driver Code

Before you debug the driver, it is a good idea to calculate the base address of driver code, as follows:

1. Run SYSGEN and issue the SHOW/DEVICE command. The resulting display lists the location in nonpaged pool at which SYSGEN loaded the driver.
2. Consult the loadmap for the driver (obtained at driver link time). Driver code resides in two program sections (PSECTs):

$$$105_PROLOGUE	driver prologue table
$$$115_DRIVER	driver code

   The locations given in the driver code listing are offsets from $$$115_DRIVER. Thus, you can calculate the base address of the driver by adding the address at which the driver was loaded to the offset associated with the PSECT $$$115_DRIVER shown in the map.

If you do not have the loadmap, consult the driver prologue table in the driver listing. Look for the address of DPT_STORE_END, which generates a 2-byte entry that terminates the DPT. To get the base address of driver code, add the address of DPT_STORE_END + 2 to the address at which the driver was loaded. You can set an XDELTA base register to the base of driver code; Section 13.8 describes this procedure.

13.6 Requesting an XDELTA Software Interrupt

Once the controller and unit initialization routines complete execution, you will need to set breakpoints in order to debug the driver. You can set a breakpoint in the driver source code by inserting calls to INI$BRK, as described in Section 13.4.

Note that, in a VMS multiprocessing system, only one processor can be in XDELTA at a time. If a processor encounters a breakpoint while another processor is in XDELTA, it too must wait until the current processor exits from XDELTA. When it does, this processor again executes the instruction that caused it to attempt to enter XDELTA. If the processor previously in XDELTA did not delete the breakpoint, this processor now enters XDELTA. If the processor previously in XDELTA did remove the breakpoint, this processor does not enter XDELTA.

You can also invoke XDELTA to set breakpoints interactively by requesting an XDELTA software interrupt.

The procedures described in Table 13–3 issue a software interrupt to a single processor at IPL 14.

On the processor requesting the XDELTA interrupt, the interrupt service routine at IPL 14 handles the interrupt by calling the routine INI$BRK, which in turn executes the first XDELTA breakpoint. XDELTA then issues this message:

```
1 BRK AT nnnnnnnn
address/NOP
```

13.6 Requesting an XDELTA Software Interrupt

In a VMS multiprocessing system, if another processor attempts to enter XDELTA at this time, it must wait until the processor currently in XDELTA exits.

Table 13–3 Requesting an XDELTA Software Interrupt

System	Boot Commands

VAX 9000	$ Ctrl/P
	>>> HALT
	>>> D/I 14 E
	>>> C

VAX 85*x*0/8700/88*x*0[1]
VAX 6000 Series[2]
VAXstation 3520/3540[3]
	$ Ctrl/P
	>>> HALT
	>>> D/I 14 E
	>>> C

VAX 8600/8650[2]	$ Ctrl/P
	>>> HALT
	>>> D/I 14 E
	>>> C

VAX 82*x*0/83*x*0
VAX–11/750
VAX–11/730[2]
	$ Ctrl/P
	>>> D/I 14 E
	>>> C

VAX–11/780
VAX–11/785
	$ Ctrl/P
	>>> HALT
	>>> DEPOSIT/I 14 E
	>>> CONTINUE

MicroVAX Systems[2]	Press and release the HALT button on the CPU control panel, or press the BREAK key (if enabled) on the console terminal. Then issue these commands:
	>>> D/I 14 E
	>>> C

[1]Note that the console prompt for the VAX 8810/8820/8830 systems is *PS-CIO-0>* and not *>>>*.

[2]These VAX systems accept only 1-character console commands.

[3]Note that command *C* for VAXstation 3520/3540 systems will only continue the primary processor. To continue all processors, use command *C/ALL*.

13.7 Examining the Vector-Jump Table

To gain familiarity with the I/O database, you might wish to look for the address of the location in the channel request block that contains a JSB instruction to the driver's interrupt service routine. You can do this at a controller initialization breakpoint because one of the inputs is the IDB address. The procedures for locating the driver interrupt service routine on non-direct-vector and direct-vector adapters follow.

Non-Direct-Vector Procedure

```
R5/ IDB-address  Q+10/ADP-address
Q+10/ vector-table-address
Q+vector-address-in-hex/ address-of-JSB-instruction-in-CRB
Q! JSB-instruction
```

Direct-Vector Procedure

```
R5/ IDB-address      Q+10/ADP-address
Q+10/ vector-table-address
Q+vector-address-in-hex+2/ address-of-JSB-instruction-in-CRB
Q! JSB-instruction
```

Finding the address of the driver's interrupt service routine at the expected vector does not guarantee that an interrupt from the device will dispatch to the driver's interrupt service routine. If the device's physical vector is set to some other address, an interrupt from the device can dispatch to some other interrupt service routine, or dispatch to an unassigned vector.

See the SYSGEN device table shown in Table 12–2 for a list of vectors. Consult Digital customer service for help with any problem similar to the one described above.

13.8 Setting an XDELTA Base Register

During a driver debugging session, you can use an XDELTA relocation register as a base from which to examine driver code and set breakpoints within the driver. Use one of the methods outlined in Section 13.5 to determine the base address of driver code, then set a relocation register by issuing the following command:

```
driver-base-address,0;X  Return
```

This command sets relocation register X0 to the base of driver code. Now you can examine offsets into the code using X0 as a base:

```
X0 + offset/ nnnnnnnn
```

or

```
X0 + offset! instruction
```

XDELTA also uses the base register to display address values in the base register plus offset format. Suppose, for example, that your driver contains the following code:

```
     50   81   90   00D3   132 10$:     MOVB   (R1)+,R0
          10   13   00D6   133          BEQL   20$
     20   50   91   00D8   134          CMPB   R0,#^A/ /
          F6   19   00DB   135          BLSS   10$
7A   8F   50   91   00DD   136          CMPB   R0,#^A/Z/
          F0   14   00E1   137          BGTR   10$
     82   50   90   00E3   138          MOVB   R0,(R2)+
          EB   11   00E6   139          BRB    10$
```

If base register 0 contains the base address of your driver, the following XDELTA dialogue is possible:

```
X0+D3,X0+E6! X0+D3/MOVB (R1)+,R0
X0+D6/BEQL     X0+E8
X0+D8/CMPB     R0,#20
X0+DB/BLSS     X0+D3
X0+DD/CMPB     R0,#7A
X0+E1/BGTR     X0+D3
X0+E3/MOVB     R0,(R2)+
X0+E6/BRB      X0+D3
```

To set breakpoints in driver code, use the following command:

```
X0 + offset;B  Return
```

To display a driver instruction and set a breakpoint, add the instruction's offset to the base register. For example:

```
X0+1C! instruction    .;B  Return
```

The last XDELTA command sets a breakpoint at the displayed location. See Section 13.10 or the *VMS Delta/XDelta Utility Manual* for a detailed discussion of XDELTA commands.

13.9 Examining the UCB, IRP, or PSL

In addition to using XDELTA to debug drivers, you can also examine the contents of the UCB and the associated IRP.

It is also useful to examine the contents of the PSL at the time of a system failure. The PSL, for example, indicates the IPL at the time. When the system fails, it prints the PSL and other register contents on the console terminal.

While the system is running, the following command can be used to examine the PSL in XDELTA:

```
RF+4/
```

The PSL location is stored in the longword following the PC.

13.10 XDELTA Commands

Table 13–4 summarizes XDELTA commands. The sections that follow this table describe the commands.

Table 13–4 XDELTA Command Summary

Command	Function
Set Display Mode	
[B	Set byte mode
[W	Set word mode
[L	Set longword mode
[I	Set instruction mode
"	Set ASCII mode
Set and Proceed from Breakpoint	
;P	Proceed from breakpoint
;B	Set/clear/display breakpoint
Open, Examine, and Close Location	
/	Open location (display contents in current mode)
!	Open location (display contents as instructions)
Return	Close current location
Crtl/J	Close current location; open next
Tab	Open location specified by current value
Crtl/3	Display previous location
Deposit in Location	
'string'	Deposit string at current location, autoincrementing the current location symbol (.). Every carriage-return and line-feed character typed will be stored. An apostrophe terminates the string.
Step, Set Location, and Execute Code	
S	Execute one instruction, step into subroutine call
O	Execute one instruction, step over subroutine call (on CALL*x*, JSB, or BSB*x* instruction)
;G	Go to location and proceed
;E	Execute command string at location
Special Symbols	
,	Field separator
Q	Last quantity displayed
=	Display value of expression; set Q

(continued on next page)

Table 13–4 (Cont.) XDELTA Command Summary

Command	Function
Special Symbols	
X*n*	Base register *n*
;X	Set base register
R*n*	Register *n*
P*n*	Processor register *n*
G	Add ^X80000000 to subsequent or preceding value
H	Add ^X7FFE0000 to subsequent or preceding value
.	Current location
Operators	
+	Add
–	Subtract
space	Add
*	Multiply
@	Shift
%	Divide
Miscellaneous	
;L	List names and locations of loaded executive images

13.10.1 Values and Expressions

All numeric values are interpreted in hexadecimal radix. Expressions are strings of alternating values and binary operators, where the first and last items in the string are always values, as in the following example:

```
G4A32 + 24 - .
```

XDELTA evaluates expressions from left to right with no precedence, and ignores trailing operators. To display the value of an expression, use the XDELTA Show Value (=) command, as follows:

Syntax

```
expression= value-of-expression
```

Type an expression followed by an equal sign (=). The expression can be composed of a series of values and operators from the set of operators listed in the command summary. XDELTA shows the value of the expression according to the current display data type. The last quantity (Q) is set to the value of the computed expression.

13.10.2 Special Symbols

XDELTA defines the following special symbols:

.	Current location; set by slash (/), exclamation point (!), and Tab operations.
Q	Last quantity displayed; you can also change this value by using the Show Value (=) command described in Section 13.10.1.
X0–XF	Base registers; used for remembering values. Set base registers by means of the Set Base Register command (;X) described in Sections 13.8 and 13.10.2.3. XDELTA, by default, stores special values in base registers X4 and X5 that help reference the process control block of the current process (see Section 13.10.2.1). Also, XDELTA initializes XE and XF with special commands that help reference page-frame numbers, as described in Section 13.10.2.2.
R0–RF	General register names.
P0–Pnn	Internal processor registers.
RF+4	PSL.
G	^X80000000; prefix for system space addresses; for example, G2E is equivalent to ^X8000002E.
H	^X7FFE0000; prefix for control region prefix; for example, H2E is equivalent to ^X7FFE002E.

13.10.2.1 Stored Base Registers

XDELTA defines two base registers useful in system debugging: X4 and X5. Base register X4 contains the address of the location that contains the address of the PCB of the current process on the current processor. Base register X5 corresponds to the global symbol SCH$GL_PCBVEC, which contains the starting address of the list of PCB slots.

13.10.2.2 Stored Command Strings

XDELTA contains two predefined command strings whose addresses are contained in base registers XE and XF. You can use these commands during general system debugging as well as driver debugging; they perform the following functions:

XE	Use the value of base register X0 as a page-frame number and display the PFN database for that page
XF	Set base register X0 to the value (PFN) in R0 and perform the same function as XE

You must initialize the stored commands to set the relocation registers they use (X6 to XD). Issue the following commands:

```
XE;E  Return
XF;E  Return
```

After executing these commands, you can use the commands stored in XE and XF to obtain the following information about a page-frame number:

- Specified physical page number (PFN)
- PFN state
- PFN type

- PFN reference count
- PFN backward link/working set list index
- PFN forward link/share count
- Page-table entry (PTE) pointer to PFN
- PFN backing store address
- Virtual block number in process swap image

13.10.2.3 Setting Base Registers

Syntax

```
address-expression,n;X  Return
```

Type an expression followed by a comma (,), a single digit between 0 and D (hexadecimal), a semicolon (;), and the letter X. XDELTA assigns the specified expression to the base register selected by *n*. XDELTA confirms that the base register is set by displaying the value deposited in the base register.

Whenever XDELTA displays an address located close to an address stored in a base register, XDELTA displays the base register identifier (Xn), followed by an offset that gives the address's location in relation to the address stored in the base register. For example, if base register 2 (X2) contains 800D046A and the address XDELTA needs to display is 800D052E, XDELTA displays X2+C4. XDELTA computes relative addresses for opened or displayed locations and addresses that are instruction operands.

XDELTA displays an address in base register plus offset format to a distance of 800_{16} from the base register. If the address falls outside this range, XDELTA displays it as a hexadecimal value.

13.10.3 Display Names and Locations of Loaded Executive Images

Syntax

;L

Use the ;L command to list the names and locations of the loaded modules of the VMS executive. If you issue the ;L command before all the executive images are loaded (for example, at an XDELTA initial breakpoint), only those images that have been loaded will be displayed.

13.10.4 Set Display Mode

Syntax

[B	Byte width
[W	Word width
[L	Longword width

[I Instruction display (using longword width)

" ASCII display (using current width)

Type a left bracket ([) followed by one of the letters B, W, or L to change the current display width to byte, word, or longword, respectively. The default value is longword. The setting remains in effect until another display mode control command is given. For example, the following command displays the least significant byte contained at the specified address and deposits the new value to that byte only.

```
address-expression [B/ old-value  new-value
```

Type a left bracket ([) followed by the letter I to change the current display mode to instruction format. This command is equivalent to the exclamation point (!) command and, similarly, is canceled by typing a slash (/) or a quotation mark ("). Instruction mode sets display mode storage units to longword values. For an example of an instruction display, see Section 13.8.

You can display contents of memory locations in ASCII characters by typing an address expression followed by a quotation mark (").

```
address-expression" old-value-in-ASCII
```

Pressing Ctrl/J displays the next location in ASCII.

The display mode remains set to ASCII until the next slash (/) or exclamation point (!) command. At this point, the display mode reverts to hexadecimal. The width remains unchanged.

13.10.5 Open, Examine, and Close Location

XDELTA provides the commands described in the following sections to open, examine, and close the specified memory locations.

13.10.5.1 Open and Display Value Command

Syntax

```
address-expression/ old-value  [new-value-expression]
```

Type an address expression followed by a slash (/) character. XDELTA displays the contents of the location (**old-value** above), followed by a space. You can change the value at the location by typing a new value and then pressing Return. If you press the Return key without preceding it with a value, the old contents remain unchanged.

The display and the value deposited default to longword hexadecimal values. The length can be changed to byte or word with the set mode commands.

A slash preceded by a null address expression uses the displayed value (Q) as the address value. This feature is convenient for following address linked chains, as follows:

```
address-expression/ old-value  / old-value  / old-value
```

13.10.5.2 Display Instruction Command

Syntax

```
address-expression! decoded-instruction
```

Type an address expression followed by an exclamation point (!). XDELTA displays the contents of memory as a VAX MACRO instruction starting with the address you specify.

XDELTA does not make any distinction between reasonable and unreasonable instructions or instruction streams; the decoding always begins at the specified address. The display instruction command does not allow you to modify the displayed location. The command sets a flag that causes subsequent close and display next or indirect location commands to perform instruction decoding. You can reset the flag with the open and display value command.

Whenever an address appears as an instruction operand, XDELTA sets the last quantity displayed (Q) to that address. XDELTA changes Q only for operands that use program counter or branch displacement addressing modes; Q is not altered for literal and register addressing modes. This feature is useful for following branches, as follows:

```
address-expression! BRW address-2   ! instruction-at-address-2
```

13.10.5.3 Close and Display Next Location Command

Syntax

```
Ctrl/J
address/old-value
```

Press Ctrl/J. XDELTA closes the current open location, then opens and displays the value in the next location, according to the current display mode.

If instruction display is the current mode, XDELTA does not deposit a value in the open location. The next location is the first location after the instruction currently displayed. If value display is the current mode, you can deposit a value into the open location. In this case, the next location is the current location, incremented by the current data width (byte, word, or longword).

13.10.5.4 Display Range Command

Syntax

```
start-addr-expression,end-addr-expression/ contents-of-start
```

or

```
start-addr-expression,end-addr-expression! contents-of-start
```

Type two address expressions separated by a comma and followed by a slash (/) or exclamation point (!). XDELTA displays the range of addresses, using the specified display mode (value or instruction). If you specify instruction display, XDELTA decodes one or more instructions. Otherwise, XDELTA displays the contents of each location in the current data type (byte, word, or longword).

13.10.5.5 Indirect Command

Syntax

Tab
address/old-value

Press Tab. XDELTA uses the last quantity displayed (Q) as an address and displays that address and its contents using the current display mode. This command opens locations in the same way as the slash (/) and exclamation point (!) commands, but prints the information on a new line and displays the address value before showing the address's contents.

13.10.5.6 Display Previous Location Command

Syntax

Ctrl/3
address/old-value

Press Crtl/3. Unless the current display mode is instruction, XDELTA decreases the location counter by the current data width, and displays the contents of the resulting location using the current data width and type. This command is ignored in instruction display mode.

13.10.6 Breakpoints

XDELTA uses the following commands to set and clear breakpoints, display a list of set breakpoints, continue from a breakpoint, and set a complex breakpoint.

13.10.6.1 Setting Breakpoints

Syntax

address-expression;B Return

Type an address followed by a semicolon (;) and the letter B, then press Return. XDELTA sets a breakpoint at the specified location and assigns it the first available breakpoint number.

Alternate Syntax

address-expression,n;B Return

Type an address followed by a comma, a single digit between 2 and 8, a semicolon (;), the letter B, and then press Return. XDELTA sets a breakpoint at the specified location and assigns it the specified breakpoint number. Breakpoint 1 is reserved for INI$BRK.

Before XDELTA executes the instruction as a breakpoint, it suspends normal instruction processing, sets a flag that causes subsequent close and display next or indirect location commands to perform instruction decoding, and displays the following message:

```
n BRK at address
address/decoded-instruction
```

You can now enter XDELTA commands. You can reset the flag that controls instruction display mode by issuing the open and display value command.

13.10.6.2 Clearing Breakpoints

Syntax

```
0,n;B  Return
```

Type zero (0) followed by a comma, a single digit between 2 and 8, a semicolon (;), the letter B, and then press Return. XDELTA clears the specified breakpoint. Never clear breakpoint 1.

13.10.6.3 Displaying Breakpoint List

Syntax

```
;B  Return
```

Type a semicolon (;) followed by the letter B. XDELTA shows the current settings of all breakpoints. For each breakpoint, XDELTA displays the following information:

- Breakpoint number
- Address at which the breakpoint is set
- Display address (for complex breakpoints; see Section 13.10.6.5)
- Command string address (for complex breakpoints)

13.10.6.4 Proceeding from Breakpoints

Syntax

```
;P  Return
```

Type a semicolon (;) followed by the letter P, and then press Return. XDELTA continues executing at the current PC.

13.10.6.5 Setting Complex Breakpoints

Syntax

```
address-expression,n,display-addr-expression,command-string-address;B  Return
```

Type an address expression followed by a comma, a single digit between 2 and 8, another address expression, and the address of a command string. The first address is the breakpoint address; the digit equals the breakpoint number. XDELTA shows the contents of the display address in the current display mode when the breakpoint is reached. The command string address specified in the last command parameter executes after automatic display.

13.10.7 Step, Set Location, and Execute Instruction Commands

The following XDELTA commands enable you to step through and execute driver code.

13.10.7.1 Loading PC and Continuing

Syntax

```
address-expression;G  Return
```

Type an address, a semicolon, and G, then press Return. XDELTA loads the address into PC and continues executing at the new PC.

13.10.7.2 Execute Instruction and Step Command

Syntax

```
S
```

Type an S. XDELTA causes one instruction to be executed, then displays the address of the next instruction and decodes that instruction.

This command also sets a flag that causes subsequent close and display next or indirect location commands to perform instruction decoding. The open and display value command resets the flag.

If the next instruction is BSBB, BSBW, JSB, CALLG, or CALLS, this command steps into the subroutine and displays the first instruction within the routine.

13.10.7.3 Step Instruction over Subroutine Command

Syntax

```
O
```

Type an O (capital letter O). XDELTA causes one instruction to be executed, then displays the address of the next instruction and decodes that instruction.

This command also sets a flag that causes subsequent close and display next or indirect location commands to perform instruction decoding. The open and display value command resets the flag.

If the next instruction is BSBB, BSBW, JSB, CALLG, or CALLS, XDELTA executes the entire subroutine and displays the instruction that immediately follows the subroutine call; this command steps over subroutines.

13.10.8 Execute String Command

Syntax

```
address-expression;E Return
```

Type an address expression followed by a semicolon, the letter E, then press Return. This command executes the ASCII commands found at the specified address expression. If you terminate the ASCII commands with a semicolon followed by the letter P, XDELTA will proceed with program execution. If you terminate the string with null (1 byte of 0), XDELTA waits for a new command.

To create command strings, open the address of the start of the string and deposit ASCII text as follows:

```
address/ old-contents 'XDELTA-command' Return
```

You can use any XDELTA command, including Return, Ctrl/J, and Tab.

To terminate the string with a null, follow the above command with

```
./ old-contents 0 Return
```

You can preassemble command strings within your experimental driver. Locate the addresses of these strings as you would any other address within your driver.

13.10.8.1 Locating Nonpaged System Patch Space

When debugging a non-Digital-supplied device driver with XDELTA, you may need to construct a complex breakpoint or store an XDELTA command string in nonpaged system patch space.

Each of the loadable images of the executive contains an area reserved as nonpaged system patch space. In each loadable image, the symbol PAT$A_NONPAGED contains a descriptor that identifies the location and size of the unused nonpaged system patch space in that image. This descriptor has the following form:

```
PAT$A_NONPAGED::
     .LONG     size-in-bytes
     .LONG     offset to patch-space-start-address
```

A suitably privileged process can access unused system patch space in any of the loadable images of the VMS executive. To determine the size of patch space and its starting address in any given loadable executive image, perform the following steps:

1 Issue the following commands to display a list of all images of the VMS executive that have been loaded into memory:

```
$ ANALYZE/SYSTEM
SDA> SHOW EXECUTIVE
SDA> EXIT
```

The XDELTA command ;L also displays a list of the loaded images.

2 Note the base address of the image whose patch space you want to use.

For example, you may have selected PROCESS_MANAGEMENT and determined its base address to be $80127C00_{16}$.

3 Determine the image value of the nonpaged system patch space descriptor (PAT$A_NONPAGED) in the selected image.

For example, to determine the image value of PAT$A_NONPAGED in PROCESS_MANAGEMENT, issue the following commands from the DCL prompt:

```
$ ANALYZE/IMAGE/OUT=TEMP.DAT SYS$LOADABLE_IMAGES:PROCESS_MANAGEMENT.EXE
$ SEARCH/WIND TEMP.DAT PAT$A_NONPAGED
```

Suppose these commands determine the image value of PAT$A_NONPAGED in SYS$LOADABLE_IMAGES:PROCESS_MANAGEMENT.EXE to be 6544_{16}.

4 Use the Patch Utility to locate PAT$A_NONPAGED in the image and examine its contents.

The following commands locate and examine PAT$A_NONPAGED in SYS$LOADABLE_IMAGES:PROCESS_MANAGEMENT.EXE:

```
$ PATCH/NOJOURNAL SYS$LOADABLE_IMAGES:PROCESS_MANAGEMENT.EXE
PATCH> EXAMINE 6544
00006544:  00000077
PATCH> EXAMINE
00006548:  00000F85
PATCH> EXIT
```

In this example, the Patch Utility output shows that there are 77_{16} bytes remaining in the nonpaged system patch space of PROCESS_MANAGEMENT and that the available patch space starts at offset $F85_{16}$ into the image.

5 Calculate the starting address of nonpaged system patch space in the selected loadable executive image by adding the offset from the descriptor to the base address of the image you determined in step 2.

For instance, the base address of nonpaged system patch space in SYS$LOADABLE_IMAGES:PROCESS_MANAGEMENT.EXE is $80127C00_{16}$ +$F85_{16}$, or $80128B85_{16}$.

13.11 Guidelines for Debugging Device Drivers

The following sections discuss errors commonly made during debugging sessions and describe additional debugging techniques.

13.11.1 Opening Device Registers in XDELTA

References to 16-bit device registers must be word instructions; references to 8-bit device registers must be byte instructions. These restrictions apply to the XDELTA EXAMINE command; therefore, be sure to set the correct mode control before examining device registers. For example, if the address of the device CSR is in R4, give the following command:

```
R4/ csr_address  [W/ csr_contents
```

13.11.2 Adjusting the Device Timeout Value

When single-stepping through driver code using XDELTA, it may be necessary to adjust the device's timeout value (as specified in the WFIKPCH or WFIRLCH macro) so that it is large enough to keep the device from timing out. When the driver debugging is complete, this value should be reset to a reasonable length of time.

13.11.3 XDELTA and System Failures

Driver errors can cause the operating system to suspend activity in such a way that you cannot invoke XDELTA. In this case, the only recourse is to induce a system failure. Follow the procedure described in the *VMS System Dump Analyzer Utility Manual*; the system will signal a fatal bugcheck.

To gain control in XDELTA following a fatal bugcheck, stop in SYSBOOT while initializing the system and clear the BUGREBOOT system parameter. The system will stop in XDELTA, thereby allowing you to examine the device UCB and other driver data to determine the driver error.

Another, more thorough, way to determine the cause of a system failure is to leave the BUGREBOOT system parameter set, allow the system to reboot, and then invoke the System Dump Analyzer (SDA) Utility to examine the condition of the I/O data structures at the time of the fatal bugcheck. The *VMS System Dump Analyzer Utility Manual* provides detailed information on fatal bugcheck stack format and how SDA can help debug a device driver.

13.12 Common Driver Errors

This section describes errors commonly made in drivers.

13.12.1 References to System Addresses

References by drivers to system addresses within the executive must use general addressing (G^) mode. For example, use

```
JSB     G^INI$BRK
```

13.12.2 Incorrect References to Device Registers

A common driver error is to access a nonexistent device register or to access the correct register with an instruction using incorrect length. On VAX systems that use direct-vector interrupts, these references cause a fatal machine check exception. On VAX systems using non-direct-vector interrupts, these references cause a UNIBUS adapter error interrupt. The system logs the adapter error and continues.

In many cases, the saved PC on the stack is the address of the instruction that caused the error. In other cases (for example, when the offending instruction is executed at IPL 31), the saved PC is not the address of this instruction but an address some number of instructions later, when the system actually services the interrupt.

13.12.3 Destroying Register Contents

Because the driver frequently calls VMS I/O routines, you must be careful to anticipate the register usage of these routines. Most VMS common I/O support routines use R0 to R3 freely. A frequent driver bug is to load a value into R3 and expect to find it intact after a call to allocate or load adapter resources.

Other VMS I/O routines write into R4. In some cases, the use of R4 is obvious; for example, IOC$REQSCHANL writes the device's CRB address into R4. In other cases, you might not anticipate the use of R4.

For example, EXE$IOFORK saves the calling code's R4 in a fork block, and then writes the device's IPL into R4. Because the normal flow of events is that an interrupt service routine restores a driver with a JSB instruction and the driver then calls EXE$IOFORK which returns to the interrupt service routine, the instructions following the JSB in the interrupt service routine can only assume R5 is still untouched. The coding sequence is as follows:

```
    MOVQ    UCB$L_FR3(R5),R3        ;Restore R3-R4.
    JSB     @UCB$L_FPC(R5)          ;Restore the driver process
    .
    .
    .
;Between these instructions, the interrupt service routine
;can make no assumptions about the contents of R0 to R4.
    .
    .
    .
    POPR    #^M<R0,R1,R2,R3,R4,R5> ;Restore interrupt registers
    REI                            ;Return from the interrupt
```

13.13 Pool Checking Mechanism

Certain system failures cannot easily be traced to a single instruction or to a single piece of kernel-mode code. If a device driver, for example, accesses memory that it has not properly allocated or continues to use memory that it has deallocated, a system failure can occur long after the driver has completed its activity. The system may crash when another operating system thread executes and attempts to use the corrupted data.

Special pool checking code in the VMS memory allocation and deallocation modules can help isolate problems of this sort reliably and quickly. In a normal VMS system, this code is disabled. For a system experiencing frequent and inexplicable failures, you can enable pool checking by setting the POOLCHECK system parameter.

When enabled, pool checking routines execute whenever pool is deallocated or allocated.[1]

On any deallocation of pool using the pool allocation routines[2], the routine fills the deallocated packet with a "free" pattern specified in the POOLCHECK system parameter. The first five longwords of the packet are not filled, but instead contain the following information:

- Forward and backward links into the free list
- Size, type, and subtype fields
- Address of the code that deallocated the packet
- Checksum

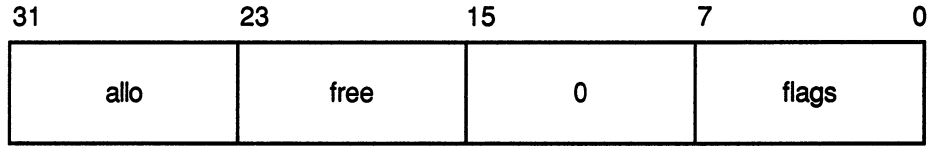
[1] The pool allocation routines (EXE$ALLOCBUF, EXE$ALLOCIRP, EXE$ALONONPAGED, COM$DEANONPAGED, EXE$DEANONPAGED, and so on) are discussed in *VMS Device Support Reference Manual*. These routines are the only means recommended by Digital for allocating pool.

[2] A small number of VMS facilities use other methods of obtaining pool. The pool allocated by these facilities is, by consequence, exempt from the pool checking mechanism.

On any allocation from pool, the routine verifies the checksum and ensures that the packet still contains the "free" pattern. If the pattern is still intact, the routine replaces the "free" pattern with an "allo" pattern, also specified in the POOLCHECK system parameter. The two patterns allow allocated, uninitialized pool to be distinguished from nonallocated pool. If the "free" pattern is not intact, the pool checking routine induces a POOLCHECK bugcheck, assuming that some code has modified the packet while it was on the free list.

Figure 13–1 illustrates the format of the POOLCHECK system parameter.

Figure 13–1 Format of the POOLCHECK System Parameter



ZK-6618-GE

The **flags** byte indicates the actions that the pool checking code should take whenever pool is allocated or deallocated. It also indicates the type of pool to be subject to checking. The following bit masks are defined:

Flag Bit	Bit Mask (hex)	Action
0	1	At deallocation, fill variable pool packets with **free** pattern.
1	2	At allocation, check packets for **free** pattern and, if the pattern is intact, fill with **allo** pattern. If not, induce POOLCHECK bugcheck.
2	4	At deallocation, fill SRPs with **free** pattern.
3	8	At deallocation, fill IRPs with **free** pattern.
4	10	At deallocation, fill LRPs with **free** pattern.
5	20	Validate the lookaside list deallocation when the associated bit (2, 3, or 4) is set. Validation ensures that the packet address is on a proper boundary before the packet is deallocated to the free list.
7	80	At deallocation, fill P1/P0-space buffers with **free** pattern. Note that the DCL lexical function F$SEARCH and the NETACP process do not function properly whenever this bit is set.

The **free** byte indicates the character to be inserted in a packet (except for its header) when it is deallocated to free pool. Select a value that will facilitate examining pool with SDA (and also could not be interpreted by the system as a valid address): for instance, an ideal value would be "d" (64_{16}), for "deallocated".

Note: If, during a single bootstrap, you enable pool checking with a certain free byte value, disable it, and then later reenable it with the same free byte value, the system may signal spurious POOLCHECK bugchecks. After pool checking has been reenabled in this manner, the pool checking mechanism may encounter an allocation of pool which only partially contains the free pattern, because of a deallocation that occurred when checking was disabled.

The **allo** byte indicates the character to be inserted in a packet (if bit 2 of the **flags** byte is set) when it is allocated. Select a value that will facilitate examining pool with SDA (and also could not be interpreted by the system as a valid address): for instance, an ideal value would be "a" (61_{16}), for "allocated."

It is possible that, when first enabled on a system, the pool checking mechanism will discover specific violations of pool allocation and deallocation protocol. In investigating subsequent crashes using SDA, you should first check the value of the global longword EXE$GL_POOLCHECK to determine whether pool checking has been enabled and, if so, which packets it has been enabled for and which patterns it is using.

One of the results of the pool checking mechanism is the occurrence of a fatal system bugcheck such as INVEXCEPTN, SSRVEXCEPT, or FATALEXCPTN whenever kernel-mode code attempts to use an address in free pool. When these exceptions signal an access violation and the **free** pattern appears as the violating address in the exception's signal array, the exception PC has been caught in the process of using deallocated pool.

The pool checking mechanism explicitly generates a POOLCHECK bugcheck for one of several reasons. A reason value is pushed onto the top of the stack as follows:

Longword Value	Reason
0	Corrupt packet
1	Bad alignment
2	Bad alignment of lookaside list element
3	Paged block partially outside paged pool
4	Nonpaged block partially outside nonpaged pool
5	IPL too high

Corrupt packet

From the top-of-stack, a reason value of 0 indicates a pool packet being allocated might have been corrupted while on the free queue. There are several causes of a corrupt packet bugcheck. You can obtain information about the crash from the contents of general registers as well as from the pool packet itself.

13.13 Pool Checking Mechanism

At the time of the crash, the following registers contain relevant information:

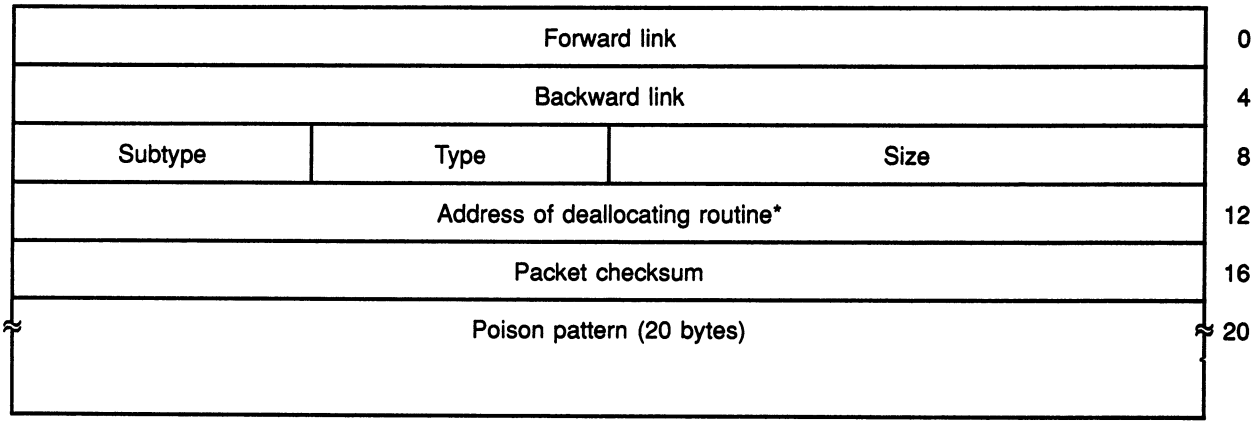
Register	Contents
R0	Allocation (**allo**) pattern
R1	Deallocation (**free**) pattern
R2	Address of packet being allocated
R3	Number of longwords remaining in packet to be checked
R4	Address in packet where the pool checking code discovered corrupted pattern
R5	Checksum, computed as the sum of the address of the packet, the deallocation pattern, the third and fourth longwords of the packet, and a longword within the system boot time quadword (EXE$GQ_BOOTIME)

Because the address of the packet is in R2, you can attempt to format R2 to see what type of structure the pool is being allocated for. The following SDA commands accomplish this:

```
SDA>  READ SYS$SYSTEM:SYSDEF.STB
SDA>  FORMAT @R2
```

If this does not identify the structure, you may obtain some information from the packet itself, as pictured in Figure 13–2.

Figure 13–2 Poisoned Pool Packet



*Note that the deallocation routine is only valid when the packet is from a lookaside list.

To determine what code may have corrupted the packet, it may be helpful to examine the contents of R4 (the address at which the pool checking routine found a corrupted pattern). If this address contains an address, it is a fair assumption that it was placed there by code that uses that address.

In addition, the routine that deallocated the packet may be a likely suspect. Bugchecks can occur if you deallocate in pieces nonpaged pool that you originally allocated as a unit. Frequently, pool is corrupted by a device driver that deallocates pool and later attempts to use the pool that it has deallocated.

Bad alignment

A top-of-stack longword value of 1 indicates that the lookaside list element being deallocated is not aligned on the boundary defined by constant EXE$C_ALCGRNMSK. The memory address to be deallocated is stored in R0.

Bad alignment lookaside list element

A top-of-stack longword value of 2 indicates that the lookaside list element being deallocated is misaligned. Specifically, a valid lookaside packet address was not supplied to the routine. At the time of the crash, the following registers contain relevant information:

Register	Contents
R0	Starting address of memory to be deallocated
R1	Size of list elements
R2	Starting address of lookaside list

Paged block partially outside paged pool

A top-of-stack longword value of 3 indicates the paged block being allocated is partially outside of paged pool. This bugcheck indicates that the memory starts in paged pool, but ends outside of paged pool. At the time of the crash, the following registers contain relevant information:

Register	Contents
R0	Starting address of block
R1	Size of block
R2	Ending address of block

Nonpaged block partially outside nonpaged pool

A top-of-stack longword value of 4 indicates the nonpaged block being allocated is partially outside nonpaged pool. This bugcheck indicates that memory being allocated starts in nonpaged pool, but ends outside of nonpaged pool. At the time of the crash, the registers shown in the table above contain the same relevant information.

IPL too high

A top-of-stack longword value of 5 indicates the IPL was above IPL$_ ASTDEL when an attempt was made to allocate or deallocate P1 space.

13.14 Detecting Driver Problems in a Multiprocessing System

When testing a new driver that has been designed to run in a VMS multiprocessing environment, it is a good idea to ensure that the system in which the driver is being tested is running the full-checking synchronization image. You can cause the full-checking synchronization image to be loaded at boot time on either a VAX uniprocessing system or multiprocessing system by the appropriate setting of the MULTIPROCESSING system parameter, as listed in Table 13–5.

Table 13–5 Settings of MULTIPROCESSING System Parameter

Value	Result
0	Loads uniprocessing synchronization image for any hardware configuration.
1	Loads full-checking synchronization image and sets multiprocessing-enabled bit (SMP$V_ENABLED in SMP$GL_FLAGS) if the hardware configuration is capable of multiprocessing and two or more processors are available; otherwise, loads uniprocessing synchronization image.
2	Loads full-checking synchronization image and sets multiprocessing-enabled bit regardless of the hardware configuration.
3	Loads streamlined synchronization image and sets multiprocessing-enabled bit if the hardware configuration is capable of multiprocessing and two or more processors are available; otherwise, loads uniprocessing synchronization image. This is the default value.

In a processing environment with the full-checking synchronization image loaded, violation of spin lock synchronization by a device driver will produce the bugchecks described in Table 13–6.

Table 13–6 Bugchecks Produced by Full-Checking Multiprocessing

SPLIPLHIGH	A processor has attempted to acquire a spin lock at an IPL higher than the IPL associated with spin lock synchronization (SPL$B_IPL). SMP$ACQUIRE (called by the LOCK and FORKLOCK macros with **condition=NOSETIPL** not specified) signals this bugcheck. A processor has attempted to acquire a device lock—not already owned by the acquiring processor—at an IPL higher than the IPL associated with device lock synchronization (SPL$B_IPL). SMP$ACQUIREL (called by the DEVICELOCK macro with **condition=NOSETIPL** *not* set) signals this bugcheck.
SPLIPLLOW	A processor has attempted to conditionally or unconditionally release a spin lock or device lock at an IPL lower than the IPL at which it originally acquired it. SMP$RELEASE and SMP$RESTORE (called by the UNLOCK and FORKUNLOCK macros) and SMP$RELEASEL or SMP$RESTOREL (called by the DEVICEUNLOCK macro) signal this bugcheck.

(continued on next page)

Table 13–6 (Cont.) Bugchecks Produced by Full-Checking Multiprocessing

SPLACQERR	A processor has attempted to acquire a spin lock while holding a higher ranked spin lock. SMP$ACQUIRE, SMP$ACQUIREL, and SMP$ACQNOIPL (called by the LOCK, FORKLOCK, and DEVICELOCK macros) signal this bugcheck.
SPLRELERR	An attempt has been made to completely release a spin lock not owned by the releasing processor. SMP$RELEASE and SMP$RELEASEL (called by the UNLOCK, FORKUNLOCK, and DEVICEUNLOCK macros) signal this bugcheck.
SPLRSTERR	An attempt has been made to conditionally release a spin lock not owned by the releasing processor. SMP$RESTORE and SMP$RESTOREL (called by the UNLOCK, FORKUNLOCK, and DEVICEUNLOCK macros when **condition=RESTORE** is specified) signal this bugcheck.

An examination of the crash dump resulting from any of these bugchecks can help locate the cause of the crash. Enter the System Dump Analyzer (SDA) and perform the following steps:

1 Issue the following command:

```
SDA> READ/EXECUTIVE SYS$LOADABLE_IMAGES
```

This command generates the symbols that correspond to locations in the loadable images that are part of the VMS executive. These symbols facilitate the interpretation of addresses that appear in the stacks and other SDA displays.

2 Issue the following command:

```
SDA> SHOW STACK
```

Trace through the current stack to determine what activities on the processor led to the acquisition or release of the spin lock. Start at high stack addresses and work towards low addresses, identifying everything on the stack, or as much of it as required to decipher what is going on.

3 Issue the following command:

```
SDA> SHOW SPINLOCK/FULL/ADDR=@R0
```

This command produces a display of information about the spin lock the executing code was trying to acquire or release, a list of PCs indicating the addresses of the latest eight acquirers or releasers of the lock, plus the PC of the last unconditional release of a set of multiply nested spin lock acquisitions. Note the acquisition IPL and the rank of the spin lock.

4 To decipher SPLIPLLO and SPLIPLHI bugchecks, compare the IPL at which the system was running at the time of the crash to that shown in the SHOW SPINLOCK display as required for acquisition of the spin lock.

5 To decipher SPLACQERR, SPLRSTERR, and SPLRELERR bugchecks, issue the following command:

```
SDA> SHOW SPINLOCK/OWNED
```

In the case of SPLACQERR bugchecks, compare the rank of the spin lock being sought with that of the currently owned spin locks.

In the case of SPLRSTERR and SPLRELERR bugchecks, determine whether the releasing processing did not in fact own the spin lock it is attempting to release.

Standard drivers seldom release spin locks and fork locks. When they do, they should be careful to use the **condition=RESTORE** argument to the UNLOCK and FORKUNLOCK macros when it is likely that the driver code is executing at the behest of other code interested in retaining the lock.

One error of which driver writers should be wary concerns the unconditional release of a spin lock or fork lock for which there exist multiple, nested acquisitions. When multiple acquisitions of a spin lock accumulate for a processor, and one of the intermediate acquirers performs an explicit release of that lock, all ownership of the lock by that processor is entirely and immediately relinquished. If at least one of the original acquisition threads still expects the lock to be held when that thread regains control, system synchronization is broken. Moreover, when the original thread that acquired the lock itself attempts to release the lock, the system crashes with an SPLRELERR because the processor no longer owns the lock being released.

If few attempts have subsequently been made by the processor to obtain or release spin locks, you should be able to find the PC of the code that last unconditionally released the spin lock in question in SHOW SPINLOCKS /FULL/ADDRESS=@R0 display. Issue the SDA command EXAMINE/INST for each of the PCs in the display, from the top to the bottom, to determine the recent history of the lock.

14 UNIBUS and Q22 Bus Device Support

This chapter provides information specific to the creation of drivers for devices attached to the UNIBUS or Q22 bus. VMS provides extensive support for UNIBUS/Q22 bus drivers, including many system routines and macros that drivers can use to accomplish a multiblock transfer to a DMA device by means of UNIBUS adapter/Q22 bus interface resources. Section 14.1 explains the functions of the UNIBUS adapter and Q22 bus interface, describing these resources in detail. Section 14.2 provides a step-by-step account of how UNIBUS/Q22 bus device drivers can use the facilities of VMS to accomplish DMA transfers.

Although the general mechanism of device interrupt dispatching, as defined by the VAX architecture and briefly described on Chapter 9, is the same for all VAX processing systems and I/O subsystems, certain implementation details differ. In that regard, Section 14.3 describes the means by which VAX hardware and the VMS operating system deliver a UNIBUS or Q22 bus device's interrupt to its driver's interrupt service routine.

14.1 Functions of the UNIBUS Adapter and Q22 Bus Interface

The UNIBUS adapter connects the UNIBUS, an asynchronous, bidirectional bus, to the backplane interconnect. The adapter performs the following functions:

- Arbitrates interrupts from UNIBUS devices according to their priority
- Delivers interrupts from UNIBUS devices to the processor
- Allows drivers to gain access to UNIBUS device registers using system virtual addresses
- Translates 18-bit UNIBUS addresses to physical addresses in main memory
- Provides a data-transfer path to randomly ordered physical addresses in main memory
- Provides buffered data transfer paths to consecutively increasing UNIBUS addresses, thus optimizing CPU-to-UNIBUS data transfers
- Permits byte-aligned buffers for UNIBUS devices requiring word-aligned buffer addresses

The Q22 bus closely resembles the UNIBUS. For MicroVAX systems with an attached Q22 bus, special processor logic implements a Q22 bus interface that similarly allows drivers access to device registers and manages device interrupts. Additional logic in the MicroVAX processor establishes a scatter-gather map that translates 22-bit Q22 bus addresses to physical addresses. However, Q22 bus systems do not implement

buffered data paths. Table 14–1 compares the UNIBUS and Q22 bus I/O subsystems of the various VAX and MicroVAX processing systems.

The protocol a VAX system uses to enable communications between its I/O bus and backplane permits its devices and device drivers to exchange data without much awareness of the intervening hardware. First of all, both the UNIBUS adapter and the Q22 bus interface provide access to device registers using an address mapping scheme that is invisible to the driver. In addition, when the configuration of the I/O interface has an impact on the control of a data transfer, the driver can generally call one of the many VMS routines that handle the details of the interface.

The functional differences between I/O adapters are irrelevant to devices that do not perform DMA transfers. A driver that performs non-DMA (programmed I/O) transfers for a device on the UNIBUS can, with no alteration, perform the same services for an equivalent device on a Q22 bus.

On the other hand, the differences between the functions of the UNIBUS adapter and the Q22 bus interface are significant to those drivers that manage DMA device operations.

Section 14.2 describes the means by which device drivers set up DMA transfers, according to any of these interfaces. If a DMA driver that drives similar devices on various VAX systems must secure some measure of machine independence, it can include some run-time conditional code that branches to appropriate routines in the driver that accomplish the machine-dependent work. See the description of the ADPDISP macro in the *VMS Device Support Reference Manual* and the sample drivers that appear in Appendix C and Appendix D for guidance.

This section discusses the functions of the UNIBUS adapter and the Q22 bus, as follows:

- The discussion of reading and writing device registers in Section 14.1.1 applies to UNIBUS and Q22 bus drivers.
- The description of mapping I/O bus addresses in Section 14.1.2 pertains only to UNIBUS and Q22 bus DMA device drivers.
- The description of buffering data transfers in Section 14.1.3 relates mainly to UNIBUS drivers, although Section 14.1.3.1 contains information relevant to Q22 bus drivers as well.

Table 14–1 Features of the UNIBUS Adapters/Q22 Bus Interfaces of VAX Systems

System	Adapter	Memory References (Physical Address)	Direct Data Path	Buffered Data Paths	Map Registers	Interrupt Dispatcher
VAX–11/780 VAX–11/785 VAX 8600 VAX 8650	UBA	30-bit (via SBI)	1, no byte-aligned transfers	15, 8-byte buffer, byte-aligned transfers, LWAE,[3] prefetch	496	Nondirect-vector
VAX–11/750	UBI	24-bit (via CMI)	1, byte-aligned transfers	3, 4-byte buffer,[2] byte-aligned transfers, LWAE,[3] no prefetch	512[4]	Direct-vector
VAX–11/730	UBA	24-bit	1, byte-aligned transfers	None	512[4]	Direct-vector
VAX 82*x*0/83*x*0 VAX 85*x*0/8700 /88*x*0	DWBUA	30-bit (via VAXBI)	1, byte-aligned transfers	5, 8-byte buffer, byte-aligned transfers, LWAE,[3] no prefetch	512[4]	Direct-vector
VAX 6000 series	DWMUA	30-bit (via VAXBI)	1, byte-aligned transfers	5, 8-byte buffer, byte-aligned transfers, LWAE,[3] no prefetch	512[4]	Direct-vector
MicroVAX 3400 /3600/3900 series VAX 4000 series	—	29-bit	1, no restrictions on data alignment[1]	None	8192	Direct-vector
MicroVAX II	—	24-bit	1, no restrictions on data alignment[1]	None	8192	Direct-vector

[1]The MicroVAX Q22 bus implementation provides no byte-offset register; so, on Q22 bus devices that are only capable of word-aligned transfers, only word-aligned transfers are possible.

[2]Buffered data paths on the VAX–11/750 only buffer four bytes of data. Because the data paths do not perform a prefetch, they can always reference longwords at random.

[3]LWAE (longword access enable) refers to the capability to reference random longword-aligned data in a bus transfer.

[4]The VMS operating system makes available only 496 of these map registers.

14.1.1 Reading and Writing Device Registers

Each I/O controller or device directly attached to a UNIBUS or Q22 bus has a control and status register (CSR) and set of data registers. These registers are assigned physical addresses in the 8Kb allocated for this purpose from the 256Kb UNIBUS address space or in the Q22 bus I/O space. Device drivers obtain the device's status and activate the device by reading and writing to these registers.

Because the VMS operating system maps this I/O space into virtual address space, a device driver can treat the addresses of device registers as identical to all other virtual addresses. The driver can read and write data to the device's register as though the device's register were a location in memory. The driver must use instructions within the restrictions described in Section 5.2.

Before a driver for a device that shares a controller can gain access to a device's registers, it must first obtain a controller channel, as described in Sections 3.4.1 and 8.3.1.

14.1.2 Map Registers

DMA devices read and write data from and to memory locations using 18-bit UNIBUS addresses or 22-bit Q22 bus addresses.

A driver that performs multiblock DMA transfers for a UNIBUS device or Q22 bus device must set up any mapping or buffering mechanisms required by the system's I/O interface. For UNIBUS DMA drivers, this involves setting up sufficient map registers and, perhaps, a buffered data path prior to the transfer. Q22 bus DMA device drivers, likewise, must allocate and fill a set of map registers.

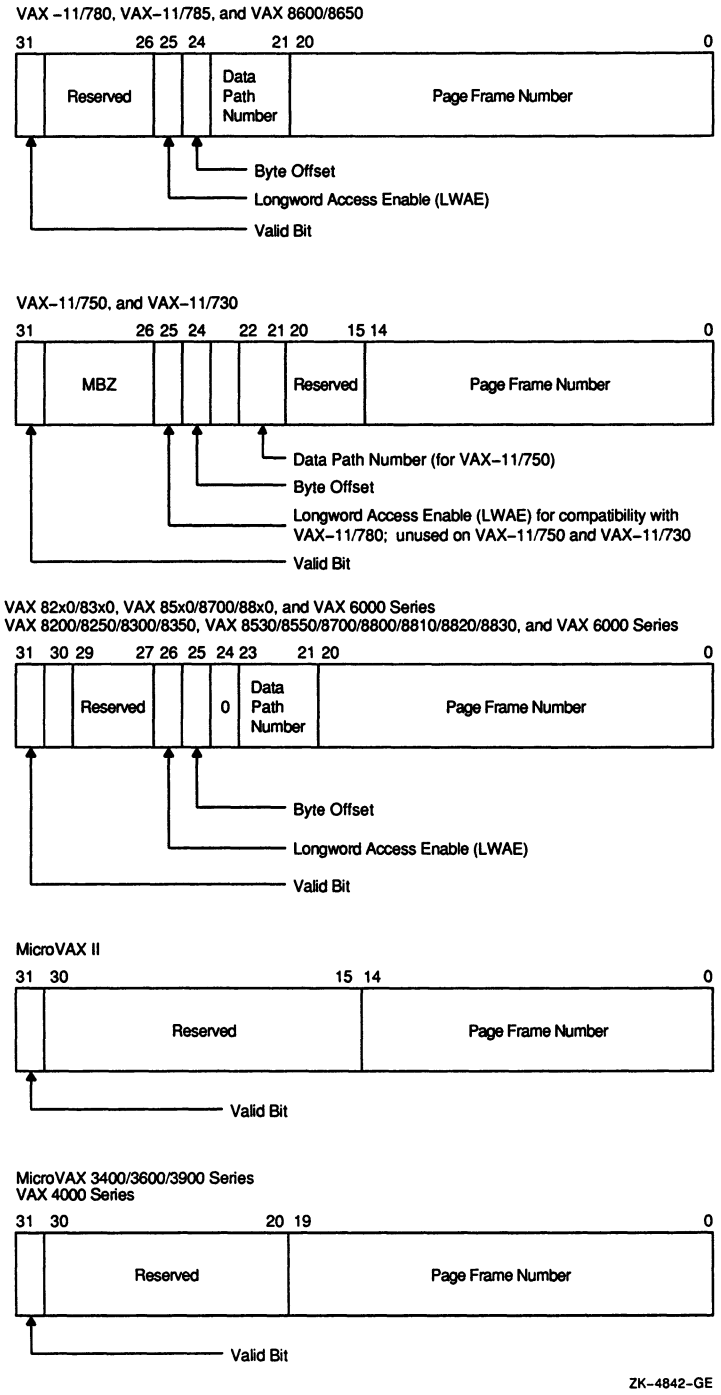
For UNIBUS and Q22 bus devices, the UNIBUS adapter and the Q22 bus interface translate the bus addresses into main memory addresses, thus allowing the operating system, I/O drivers, and UNIBUS devices to access the same physical address space. DMA devices connected to either a UNIBUS or a Q22 bus can access a block of memory indirectly by means of the scatter-gather map supplied by the UNIBUS adapter or MicroVAX processor, respectively. The map registers provided allow the device to access scattered, physical memory addresses as contiguous, physical addresses in I/O space.

When a device driver performs a DMA transfer, it allocates map registers and a buffered data path (an option available to devices on the UNIBUS of some VAX systems), and sets up the transfer by means of the device's registers. The device then accesses memory directly by means of the I/O bus, transferring all the data requested. When the transfer is complete, the device notifies the driver by requesting an interrupt.

Consider a buffer, for example, that consists of virtual pages 400, 401, 402, and 403, which are physical pages 1003, 204, 1190, and 240, respectively. For a UNIBUS or Q22 bus device to access this buffer, the driver requests four map registers, then places the physical addresses of these pages in the map registers. A field in each map register identifies the page-frame

number corresponding to the UNIBUS space or Q22 bus space address that the map register represents (see Figure 14–1).

Figure 14–1 UNIBUS and Q22 Bus Map Registers



Assume the driver has allocated four map registers, 127 through 130. The driver loads them as follows:

Map Register	Contents
127	1003
128	204
129	1190
130	240

Note that the VMS routine the driver calls to allocate map registers automatically allocates an additional map register (register 131 in this case). The map register loading routine clears this register in order to prevent a runaway DMA transfer.

The device and the UNIBUS can transfer data into or out of these physical pages without intervention by the driver. The device requests an interrupt only when all the data in these four pages has been transferred.

Generally, a map register exists for each page of I/O space. Because the UNIBUS address space consists of 256K of memory, minus the 8Kb reserved for device control registers, 496 map registers are available for UNIBUS DMA transfers. Q22 bus DMA devices can use up to 8192 of the map registers that correspond to the 4MB of Q22 bus I/O space.

Drivers call VMS routines to fill as many map registers with valid page-frame numbers as needed for a DMA transfer. The DMA device puts an address on the I/O bus. The UNIBUS adapter or Q22 bus interface receives the address and translates it using the following information (see Figures 14–2 and 14–3):

- In *UNIBUS addresses*, the 9-bit UNIBUS page address field (bits 9 through 17 of the UNIBUS address) identifies the UNIBUS adapter map register.

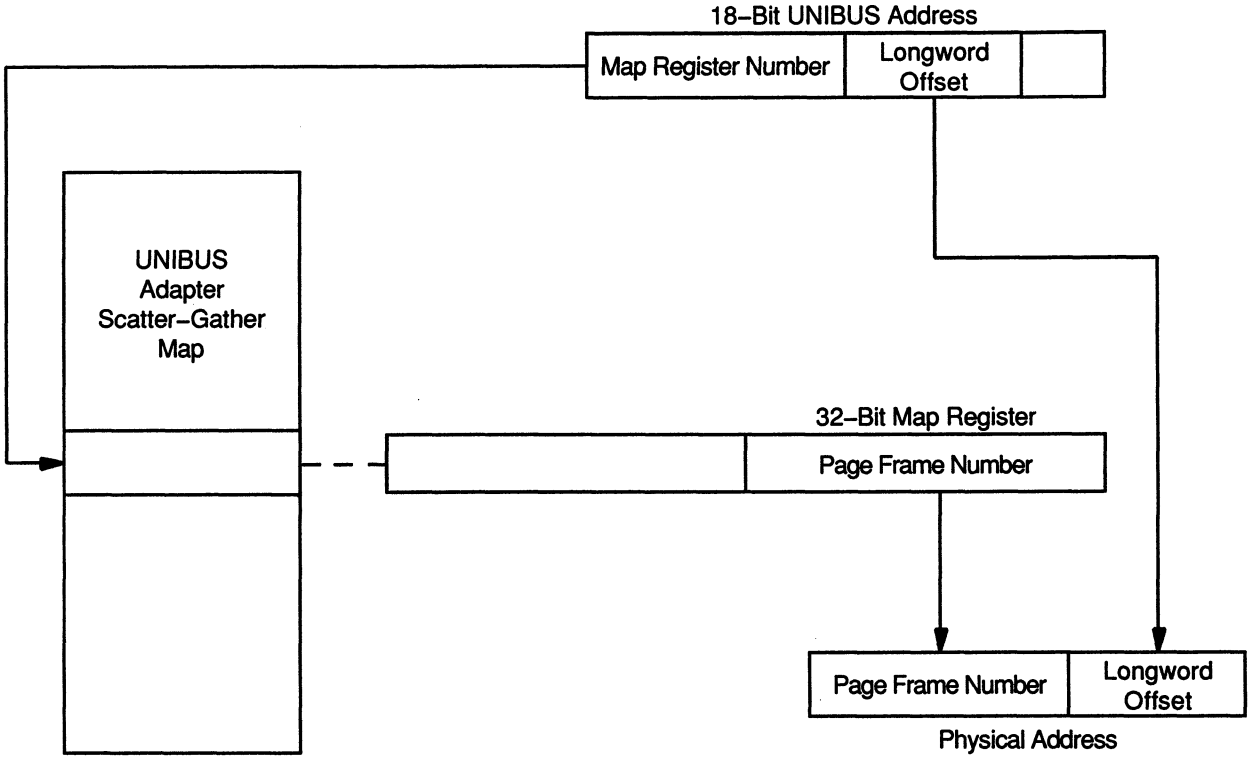
  In *Q22 bus addresses*, the 13-bit Q22 bus page address field (bits 9 through 22 of the Q22 bus address) identifies the map register.

- The page-frame-number (PFN) field in the map register specifies the high-order bits of the physical address. (The PFN field is 15 bits long for the MicroVAX II, VAX–11/750, and VAX–11/730; 20 bits long for the MicroVAX 3400/3600/3900 and VAX 4000 series systems, and 21 bits long for other VAX systems.)

- From *UNIBUS addresses*, bits 2 through 8 map to bits 0 through 6 of the physical address.[1] The resulting physical address locates the longword that is the target of the transfer.

[1] The disposition of the lowest two bits of the UNIBUS address depends on the VAX system. For instance, the VAX–11/780 uses them to construct a byte-selection mask and function to be transmitted across UNIBUS lines that modify the I/O transaction.

Figure 14–2 Mapping a UNIBUS Address to a Physical Address



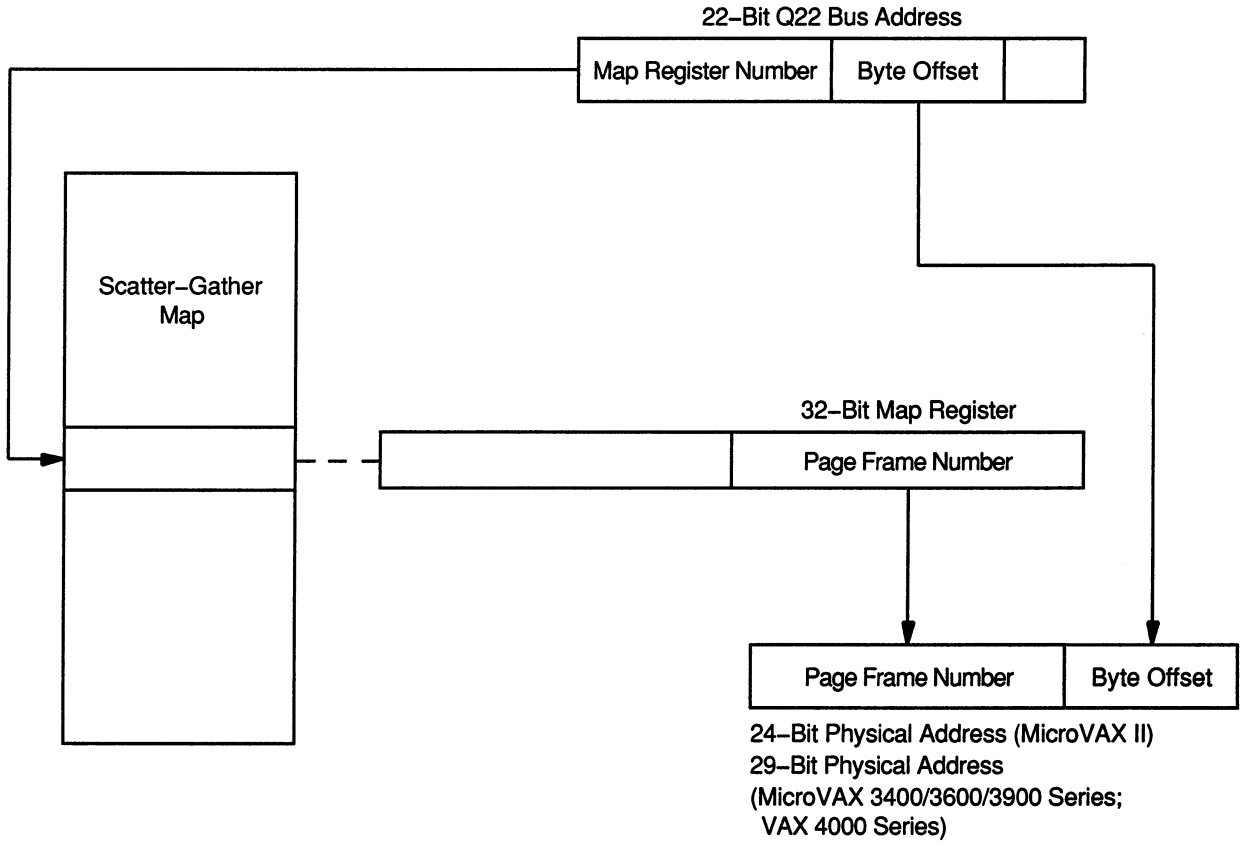
From *Q22 bus addresses* (Figure 14–3), bits 0 through 8 map to bits 0 through 8 of the physical address. The resulting physical address locates the byte that is the target of the transfer.

Each UNIBUS adapter or Q22 bus map register also contains a bit called the map-register valid bit. The UNIBUS adapter or Q22 bus interface tests this bit every time the map register is used. If the bit is not set, the UNIBUS adapter or Q22 bus interface aborts the transfer. This bit is clear whenever the register is not mapped to a physical address.

14.1.3 UNIBUS Adapter Data Transfer Paths

The UNIBUS adapter sends data through one of several data paths for UNIBUS devices performing DMA transfers. One data path, the **direct data path** (DDP), allows UNIBUS transfers to randomly ordered physical addresses. The direct data path maps each UNIBUS transfer to a backplane interconnect transfer. Thus, a single word or byte of data is transferred for each backplane interconnect operation.

Figure 14–3 Mapping a Q22 Bus Address to a Physical Address



The remaining data paths, the **buffered data paths** (BDPs), allow devices on the UNIBUS to transfer more efficiently than through the direct data path. The buffered data paths store UNIBUS data so that multiple UNIBUS transfers result in a single backplane interconnect transfer.

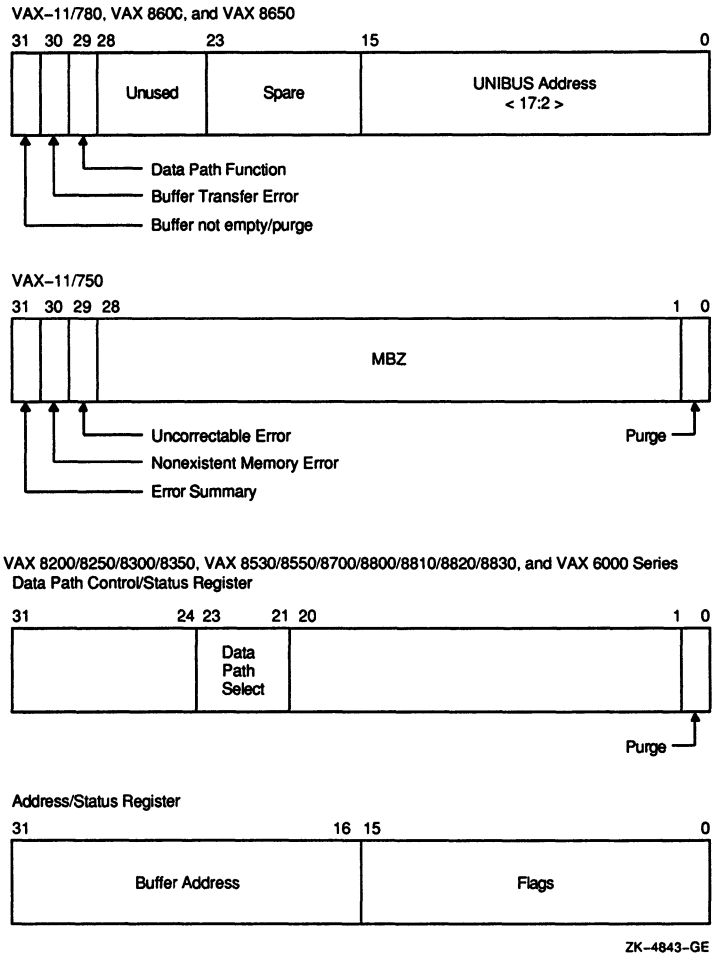
When a UNIBUS device begins a DMA transfer by placing an address on the UNIBUS, the UNIBUS adapter not only performs address mapping but also provides the number of the data path to be used for the transfer (see Figure 14–1). Each UNIBUS adapter map register contains a field that describes the data path. Data path 0 is the direct data path; the other data paths are the buffered data paths. The UNIBUS data path registers of the various VAX systems are pictured in Figure 14–4.

The following sequence describes a UNIBUS-device DMA transfer.

1 The UNIBUS device puts an address on the UNIBUS.

2 The UNIBUS adapter locates the UNIBUS adapter map register that corresponds to the UNIBUS address.

Figure 14–4 UNIBUS Data Path Registers



3 The UNIBUS adapter verifies that the map register has the map-register valid bit set.

4 The UNIBUS adapter maps the UNIBUS address to a physical address.

5 The UNIBUS adapter extracts the number of the data path to be used for the transfer from the map register.

6 The UNIBUS adapter translates the UNIBUS function to a backplane interconnect function by reading the UNIBUS control lines.

7 Based on the UNIBUS function indicated by the UNIBUS control lines (DATI, DATIP, DATO, or DATOB), the UNIBUS adapter starts appropriate UNIBUS and backplane interconnect operations to transfer data between the UNIBUS device and memory.

14.1.3.1 Direct Data Path

Since the direct data path performs a backplane interconnect transfer for every I/O bus transfer, it can be used by more than one UNIBUS or Q22 bus device at a time. The UNIBUS adapter or Q22 bus interface arbitrates among devices that wish to use the direct data path simultaneously. The device driver is unaffected by this arbitration.

The direct data path is less efficient than a buffered data path because each I/O bus transfer cycle corresponds to a backplane interconnect cycle. One word or byte is transferred for each backplane interconnect cycle. On some hardware configurations, the direct data path is unable to transfer a word of data to an odd-numbered physical address. Therefore, an FDT routine for a DMA device that uses the direct data path should check that the specified buffer is on a word boundary.[2]

The Q22 bus systems only employ a direct data path. A UNIBUS device driver may choose to use a direct data path rather than a buffered data path to perform the following functions:

- Execute an interlock sequence to the backplane interconnect (DATIP-DATO/DATOB)
- Transfer to randomly ordered addresses instead of consecutively increasing addresses
- Mix read and write functions

The direct data path is the simplest data path to program. Since the direct data path can be shared simultaneously by any number of I/O transfers, the device driver does not need to call a VMS routine to allocate the data path. Instead, the driver performs the following actions:

1 Uses the REQMPR macro to allocate a set of map registers (or the REQALT macro to allocate a set of Q22 bus alternate map registers (registers 496 to 8191)).

2 Uses the LOADUBA macro (or LOADALT macro) to load the map registers with physical address map data and, for UNIBUS devices, the number of the direct data path (0). The VMS routines called in the expansion of these macros (IOC$LOADUBAMAP and IOC$LOADALTMAP respectively) also set the valid bit in every map register except the last, which remains invalid to prevent a runaway transfer.

3 Loads the starting address of the transfer in a device register.

4 Loads the transfer byte or word count in a device register.

5 Sets bits in the device CSR to initiate the transfer.

[2] The MicroVAX implementation of the Q22 bus provides no byte-offset register. As a result, for Q22 bus devices that are only capable of word-aligned transfers, only word-aligned transfers are possible.

14.1.3.2 Buffered Data Paths

When a buffered data path is used, the UNIBUS adapter transfers data much more efficiently between the UNIBUS and the backplane interconnect than when a direct data path is used. It accomplishes this by decoupling the UNIBUS transfer from the backplane interconnect transfer. The buffered data path allows the UNIBUS adapter to read or write multiple words of data in a transfer, and buffer the unrequested portions of the data in a UNIBUS adapter buffer. Thus, several UNIBUS read functions can be accommodated with a single backplane interconnect transfer.

Q22 bus systems do not employ buffered data paths. The writer of a UNIBUS device driver may choose to use a buffered data path rather than a direct data path to perform the following functions:

- Faster DMA block transfers to or from consecutively increasing UNIBUS addresses
- Word-oriented block transfers that begin and end on an odd-numbered byte of memory; note, however, that these transfers can be quite slow because the UNIBUS adapter might need to perform multiple transfers to complete a one-word transfer
- 32-bit data transfers from random longword-aligned physical addresses

A single buffered data path cannot be assigned to more than one active transfer at a time. When a driver fork process is preparing to transfer data to or from a UNIBUS device on a buffered data path, it performs a sequence of steps similar to those performed by a driver that uses the direct data path, with the exception that it uses a macro that calls a VMS routine that allocates a free buffered data path. The following are among the actions of the driver fork process:

1. Uses the REQMPR macro to allocate a set of map registers.
2. Uses the REQDPR macro to allocate a free buffered data path.
3. Uses the LOADUBA macro to load the map registers with physical address mapping data and the number of the allocated buffered data path. The VMS routine called in the expansion of the LOADUBA macro (IOC$LOADUBAMAP) also sets the valid bit in every map register except the last, which remains invalid to prevent a runaway transfer.
4. Loads the starting address of the transfer in a device register.
5. Loads the transfer byte or word count in a device register.
6. Sets bits in the device CSR to initiate the transfer.

The UNIBUS adapter hardware of certain VAX systems normally restricts buffered data paths to referring only to consecutively increasing UNIBUS addresses. Through a special mode of operation, these UNIBUS adapters can also refer to 32-bit data at randomly-ordered, longword-aligned locations in physical memory. Other systems do not impose this restriction. In order for a device driver to run on both types of systems, it must observe three rules:

- All transfers within a block must be of the same function type (DATI or DATO/DATOB).
- Buffered data paths must always transfer data to consecutively increasing addresses.
- To reference 32-bit data at random, longword-aligned locations in physical memory, the longword-access-enable bit (LWAE) must be set.

A buffered data path stores data from the UNIBUS in a buffer until multiple words of data have been transferred (except in longword-aligned, 32-bit, random-access mode as discussed in Section 14.1.3.5). Then, the UNIBUS adapter transfers the contents of the buffer to the appropriate physical address in a single backplane interconnect operation. The procedure for a UNIBUS write operation that transfers data from a device to memory is broken into individual steps.

1 The UNIBUS device transfers one word of data to the buffered data path.

2 The UNIBUS adapter stores the word of data and completes the UNIBUS cycle.

3 The UNIBUS adapter sets the buffer-not-empty flag in the buffered data path to indicate that the buffer contains valid data.

4 The UNIBUS device repeats the first three steps until the buffer is full.

5 When the UNIBUS device addresses the last byte or word in the buffer, the UNIBUS adapter recognizes a complete data-gathering cycle.

6 The UNIBUS adapter requests a write function on the backplane interconnect to write the data from the buffered data path to memory.

7 When the backplane interconnect transfer is complete, the buffered data path clears its flag to indicate that the buffer no longer contains valid data.

The procedure for a UNIBUS read operation that transfers data from main memory to a device varies according to the type of UNIBUS adapter. Those adapters that can perform a prefetch function complete UNIBUS reads from memory more quickly than those that cannot. The prefetch feature accomplishes this improved performance by automatically filling the data path buffer after the buffer's contents are transferred to the UNIBUS.

The following paragraphs discuss the UNIBUS read operation with and without the prefetch function. Device drivers that adhere to the conventions outlined in this manual will execute properly whether or not the device is associated with a UNIBUS adapter that is capable of prefetches.

1 The UNIBUS device initiates a read operation from a buffered data path.

2 The UNIBUS adapter checks to see if its buffers contain valid data.

3 If the buffers do not contain valid data, the buffered data path initiates a read function to fill the buffers with data from main memory. The transfer completes before the UNIBUS adapter begins a UNIBUS transfer.

4 The UNIBUS adapter transfers the requested bytes to the UNIBUS. Bytes of data that were not transferred to the UNIBUS remain in the buffer.

5 The UNIBUS adapter sets the buffer-not-empty flag in the buffered data path to indicate that the buffer contains valid data.

6 When the UNIBUS device empties the buffers of the buffered data path with a UNIBUS read function that accesses the last word of data, the buffered data path clears the buffer-not-empty flag to indicate that the buffer no longer contains valid data.

7 The buffered data path then initiates a read function to prefetch data from memory.

8 When the prefetch is complete, the buffered data path sets the buffer-not-empty flag to indicate that the buffers now contain valid data.

The prefetch might attempt to read data beyond the address mapped by the final map register. To avoid referring to memory that does not exist, the VMS routines that allocate and load map registers always allocate one extra map register and clear the map-register-valid bit before initiating the transfer. When the UNIBUS adapter notices that the map register for the prefetch is invalid, the UNIBUS adapter aborts the prefetch without reporting an error.

A UNIBUS read function without prefetch includes the following steps:

1 The UNIBUS device initiates a read operation from a buffered data path.

2 The buffered data path checks to see if its buffers contain valid data.

3 If the buffers do not contain valid data, the buffered data path initiates a read function to fill the buffers with data. The transfer completes before the UNIBUS adapter begins a UNIBUS transfer.

4 The buffered data path transfers the requested bytes to the UNIBUS. Bytes of data that were not transferred to the UNIBUS remain in the buffer.

14.1.3.3 Byte-Offset Data Transfers

The UNIBUS adapter has a byte-offset register; thus, words that are not word-aligned can be transferred to and from any device on the UNIBUS regardless of whether the device supports non-word-aligned transfers.

Some UNIBUS devices are restricted to transferring integral words of data in word-aligned UNIBUS addresses. The buffered data paths allow these devices to perform transfers to memory that begins and ends on an odd-byte address. A byte-offset bit in the map registers indicates byte-aligned data to the hardware. If the bit is set, the hardware increments physical addresses. A VMS subroutine that loads map registers determines

whether the data is word- or byte-aligned and sets the byte-offset bit accordingly.

14.1.3.4 Purging a Buffered Data Path

Because prefetches can read more data from memory than the UNIBUS device wishes to read, driver fork processes must ask the UNIBUS adapter to purge the buffered data path when a transfer is complete. In addition, a transfer from a device to the backplane interconnect can complete with some data left in the buffer. The driver must purge the data path to complete the transfer.

The purge guarantees that the data is not transferred to the next user of the buffered data path. The driver fork process performs the purge by calling a standard VMS routine that performs two functions:

- Tells the hardware to purge the buffered data path register owned by the fork process. For a UNIBUS read function, the adapter simply clears the buffer-not-empty flag. For a UNIBUS write function, the adapter transfers any data left in the data path buffer to VAX memory, then clears the flag.
- Notifies the driver fork process of any error that occurs during the purge.

The data path must be purged before the driver releases map registers or the buffered data path register.

14.1.3.5 Longword-Aligned, 32-Bit, Random-Access Mode

Another method of transferring data over a buffered data path is the use of longword-aligned, 32-bit, random-access mode. This mode essentially prevents the UNIBUS prefetch operation, thereby allowing a device that reads data from or writes data to memory to reference longword-aligned locations in memory at random, in longword multiples.

To transfer data in the longword-aligned, 32-bit, random-access mode, the driver fork process sets the longword-access-enable bit (VEC$V_LWAE) in the channel request block (CRB) prior to loading the map registers. The UNIBUS device can then perform a read (DATI) or write (DATO) function.

For a UNIBUS read operation that transfers data from main memory to a device, the function occurs as follows:

1. The driver fork process initiates a read function on the UNIBUS device.
2. The UNIBUS adapter clears the buffer-not-empty flag in the assigned buffered data path.
3. The UNIBUS adapter requests a read-from-memory operation on the backplane interconnect.
4. The UNIBUS adapter stores the longword of data in the buffered data path and sets the buffer-not-empty flag.
5. The UNIBUS adapter completes two UNIBUS read operations to transfer two words of data.

For a UNIBUS write operation that transfers data from a device to main memory, the function occurs as follows:

1 The driver fork process initiates a write function on the UNIBUS device.

2 The UNIBUS adapter clears the buffer-not-empty flag in the assigned buffered data path.

3 The UNIBUS adapter completes two write operations to transfer two words of data from the UNIBUS device.

4 The UNIBUS adapter stores the longword of data in the data path's buffer and sets the buffer-not-empty flag.

5 The UNIBUS adapter initiates a backplane interconnect write operation.

6 When the backplane interconnect write operation is complete, the UNIBUS adapter clears the buffer-not-empty flag.

To ensure that random-access mode works correctly regardless of the VAX system involved, the writer of a device driver should ensure that a device assigned to a buffered data path does not repeatedly address the same longword. On certain systems, a UNIBUS device that polls a single longword, waiting for data, will constantly be returned the same data.

14.2 Writing Driver Code for UNIBUS/Q22 Bus DMA Transfers

A driver performing DMA transfers over the UNIBUS or Q22 bus must take I/O bus operation into consideration. The VMS operating system and the I/O database manage the map registers and data path resources of the I/O adapter for device drivers.

The I/O database contains an adapter control block (ADP) that describes the I/O adapter. This block contains allocation information for the map registers; for UNIBUS adapters, the ADP also contains similar information for data paths.

The ADP also contains the virtual address of the adapter's configuration register. All the adapter's other registers are located at fixed offsets from the configuration register. The VMS adapter-handling routines modify the adapter's map registers and data-path register according to requests from the driver fork process.

In general, a driver fork process does not directly access the ADP. Instead, a driver calls VMS routines that perform adapter-related services, such as the following:

- Allocating a buffered data path
- Allocating map registers or alternate map registers
- Loading map registers or alternate map registers
- Deallocating map registers or alternate map registers

- Purging a buffered data path
- Deallocating a buffered data path

The critical responsibility of device drivers that actively compete for such shared I/O adapter resources as map registers and data paths is to ensure the synchronized access of adapter resources. Drivers that share these resources must execute at the same fork IPL. In a VMS multiprocessing system, they must additionally contend for the same fork lock. A given driver code thread that must attempt to access its fork database can only do so if suitably synchronized.

The system creates a driver fork process by calling the start-I/O routine in a device driver. The fork process takes some or all of the following steps to initiate an I/O transfer to or from a device on a UNIBUS or Q22 bus.

Operation	Applicable to
Requests buffered data path	UNIBUS
Requests map registers	UNIBUS or Q22 bus
Requests alternate map registers	Q22 bus
Loads map registers	UNIBUS or Q22 bus
Loads alternate map registers	Q22 bus
Calculates starting bus address	UNIBUS or Q22 bus
Activates device	UNIBUS or Q22 bus
Waits for interrupt	UNIBUS or Q22 bus

When a hardware interrupt indicates that the I/O transfer is complete, the driver fork process checks the success or failure of the transfer. The driver then concludes with the following steps:

Operation	Applicable to
Purges data path	UNIBUS or Q22 bus
Releases buffered data path	UNIBUS
Releases map registers	UNIBUS or Q22 bus
Releases alternate map registers	Q22 bus

Because of the different requirements of DMA transfers on different VAX and MicroVAX systems, a driver must contain some run-time conditional code in order to function for equivalent UNIBUS and Q22 bus devices. The DLDRIVER code example in Appendix C is one driver that supports the RL11 on the UNIBUS and the RLV11 on the Q22 bus.

Regarding the material presented in this section, UNIBUS driver writers should read Sections 14.2.1 through 14.2.7.3. Q22 bus driver writers should read Section 14.2.1.3 and Sections 14.2.2 through 14.2.7.3.

14.2.1 Selecting and Requesting a Data Path

DMA device drivers for certain VAX systems can elect to request the use of a UNIBUS adapter buffered data path to accelerate data transfers (as described in Section 14.1.3). Other VAX processing systems, such as MicroVAX systems and the VAX–11/730, provide no buffered data paths for data transfers. The descriptions of the direct data path in the following sections apply to drivers written for devices in those systems.

14.2.1.1 Requesting a Buffered Data Path

Some VAX systems allow UNIBUS drivers to request temporary or permanent allocation of a buffered data path (see Table 14–1). After the driver fork process gains access to the controller (see Section 8.3.1), it requests a buffered data path by invoking the VMS macro REQDPR. REQDPR calls a VMS routine named IOC$REQDATAP that locates the ADP. To do this, IOC$REQDATAP uses a series of pointers that begins in the current unit control block (UCB), as follows:

UCB → CRB → ADP

IOC$REQDATAP performs the following services:

1 Tests the path-lock bit (VEC$V_PATHLOCK) in the data-path number field of the channel request block (CRB$L_INTD+VEC$B_DATAPATH). If the device has a permanent data path allocated to it, IOC$REQDATAP simply returns.

2 Determines which data paths are available by examining the data path allocation information in the ADP (ADP$W_DPBITMAP).

3 Allocates the first free data path to the driver by inserting its number in the data path field of the CRB (CRB$L_INTD+VEC$B_DATAPATH) and indicating in the ADP that the data path is in use (by clearing the appropriate bit in ADP$W_DPBITMAP).

4 Returns control to the driver fork process.

If no data path is available, IOC$REQDATAP saves driver context (R3, R4, and PC) in the UCB fork block and inserts the address of the fork block, which is also the address of the UCB and the contents of R5, in the ADP's data-path wait queue. The driver fork block remains in the queue until both of the following conditions are met:

- A data path is available.
- The driver fork block is the next entry in the data-path wait queue.

When these conditions are met, the VMS routine IOC$RELDATAP allocates the data path to the suspended driver and reactivates the driver fork process.

14.2.1.2 Requesting a Permanent Buffered Data Path

A device driver can permanently allocate a buffered data path in its unit initialization routine. Instead of using the REQDPR macro, however, the unit initialization routine should perform the following steps:

1. Issue the FORK macro to drop IPL to fork IPL. The VMS fork dispatcher causes the following steps to be performed at fork IPL under the ownership of the required fork lock in a VMS multiprocessing system. (The consequences of forking in a unit initialization routine are discussed at length in Section 11.1.5.)
2. Test the path-lock bit (VEC$V_PATHLOCK) in the data-path-number field of the CRB (CRB$L_INTD+VEC$B_DATAPATH) to ensure that a data path is not already allocated for this device.
3. Call the subroutine IOC$REQDATAPNW to allocate the data path as follows:

   ```
   JSB  G^IOC$REQDATAPNW
   ```

   If IOC$REQDATAPNW successfully allocates the data path, it stores the number of the data path it obtained in the CRB at CRB$L_INTD+VEC$B_DATAPATH and returns with the low-order bit set in R0 (SS$_NORMAL). If it cannot allocate a data path, IOC$REQDATAPNW does *not* create a fork process to wait for one to become available. Instead, it returns to the unit initialization routine with the low-order bit clear in R0.
4. If the data path has been successfully obtained, set the path-lock bit (VEC$V_PATHLOCK) in the CRB at CRB$L_INTD+VEC$B_DATAPATH.

The driver-loading procedure calls the unit initialization routine for each unit that the driver serves. A unit initialization routine that contains the code described previously will permanently allocate one buffered data path for each CRB associated with the driver, which is one path for each controller that the driver serves.

Because some VAX systems have few buffered data paths (refer to Table 14–1), device drivers running in these systems must limit their allocation of permanent buffered data paths. For example, if the drivers loaded on a VAX–11/750 permanently allocated all three of the system's buffered data paths, none would remain for normal system operations. As a result, I/O transfers requiring a buffered data path would wait forever.

14.2.1.3 Requesting the Direct Data Path

Because the UNIBUS adapter or Q22 bus interface arbitrates among devices that wish to use the direct data path and initializes the data path field in the CRB (CRB$L_INTD+VEC$B_DATAPATH) to 0 (0 = direct data path), drivers are not required to invoke the REQDPR macro to request the direct data path.

Some VAX systems, such as the VAX–11/780 or VAX 8600, do not permit byte-offset transfers on the direct data path (see Table 14–1). Because the UNIBUS itself is word-oriented, such a system must ensure that the data buffer is aligned on a word boundary for word-aligned devices.

14.2.1.4 Mixed Use of Direct and Buffered Data Paths

A UNIBUS device driver can use the buffered data path for certain operations, then use the direct data path for other operations. To accomplish this task, the driver should allocate a buffered data path for buffered I/O. When the operation is completed, the driver should then purge and release the buffered data path. The release automatically resets the data path number to zero, which signifies a direct data path. When the driver has finished using the direct data path, it should purge it (but not release it). (A purge of the direct data path is a NOP and always yields success.)

14.2.2 Requesting Map Registers

The UNIBUS adapter and Q22 bus interface allow UNIBUS and Q22 bus drivers, respectively, to allocate map registers as needed or to allocate them permanently.

14.2.2.1 Allocating Map Registers

After the driver fork process gains access to the controller (see Section 8.3.1), it can request a set of adapter map registers (registers 0 through 495) by invoking the VMS macro REQMPR. This macro calls the routine IOC$REQMAPREG. IOC$REQMAPREG calculates the number of map registers needed for a transfer, allocating one map register for each full or partial page of the buffer (based on the values of UCB$W_BCNT and UCB$W_BOFF). In addition, it reserves an additional map register to be marked invalid to stop a potential runaway transfer and inhibit prefetches from the page past that in which the end of the buffer resides. Finally, IOC$REQMAPREG may allocate one more extra map register to ensure that an even number of map registers is allocated.

The procedure for allocating map registers is similar to that used to allocate a buffered data path. First, IOC$REQMAPREG locates the ADP from a series of pointers that begins with the current UCB, as follows:

UCB → CRB → ADP

Then, the routine examines the map-register-allocation information to locate the required number of contiguous map registers. If the registers are not currently available, IOC$REQMAPREG saves the driver context (R3, R4, and PC) in the UCB fork block and inserts the fork block's address (same as UCB address and the contents of R5) in the standard-map-register wait queue.

When the map registers are available, IOC$REQMAPREG allocates them and adjusts the appropriate information about the allocation of map registers in the ADP. IOC$REQMAPREG then writes the number of the first map register and the number of map registers allocated into the CRB and returns control to the driver fork process.

VMS supplies a similar macro (REQALT) and routine (IOC$REQALTMAP) that Q22 bus drivers use to allocate a set of alternate map registers (registers 496 through 8191). REQALT and IOC$REQALTMAP perform the allocation in the same manner as REQMPR and IOC$REQMAPREG. Note that, although VMS records the allocation of standard and alternate

map registers in separate areas of the ADP and CRB, IOC$REQMAPREG and IOC$REQALTMAP use the same UCB fields to calculate the number of map registers required for a transfer.

14.2.2.2 Permanently Allocating Map Registers

A device driver can allocate a permanent set of map registers with code in its unit initialization routine. The number of map registers permanently allocated must be sufficient for the largest possible transfer and must include an extra map register to be marked invalid to prevent a runaway transfer.

A unit initialization routine performs the following steps to permanently allocate a set of map registers:

1. Issue the FORK macro to drop IPL to fork IPL. The VMS fork dispatcher causes the following steps to be performed at fork IPL under the ownership of the required fork lock in a VMS multiprocessing system. (The consequences of forking in a unit initialization routine are discussed at length in Section 11.1.5.)
2. Test the map-lock bit (VEC$V_MAPLOCK) in the CRB (CRB$L_INTD+VEC$W_MAPREG) to ensure that map registers are not already allocated for this device.
3. Load the number of map registers required into R3.
4. Call the VMS routine IOC$ALOUBAMAPN with a JSB instruction:

   ```
   JSB  G^IOC$ALOUBAMAPN
   ```

   If IOC$ALOUBAMAPN successfully allocates the map registers, it stores the number of map registers allocated and the number of the first of the allocated map registers at CRB$L_INTD+VEC$B_NUMREG and CRB$L_INTD+VEC$W_MAPREG, respectively. It then returns with the low-order bit set in R0. Otherwise, it returns with the low-order bit of R0 clear.
5. If map registers have been successfully allocated, set the map-lock bit in the CRB (VEC$V_MAPLOCK in CRB$L_INTD+VEC$W_MAPREG).

Q22 bus drivers perform the following steps to allocate a permanent set of alternate map registers (registers 496 through 8191):

1. Issue the FORK macro to drop IPL to fork IPL. The VMS fork dispatcher causes the following steps to be performed at fork IPL under the ownership of the required fork lock in a VMS multiprocessing system. (The consequences of forking in a unit initialization routine are discussed at length in Section 11.1.5.)
2. Test the alternate-map-lock bit (VEC$V_ALTLOCK) in the CRB (CRB$L_INTD+VEC$W_MAPALT) to ensure that alternate map registers are not already allocated for this device.
3. Load the number of alternate map registers required into R3.
4. Call the VMS routine IOC$ALOALTMAPN with a JSB instruction:

   ```
   JSB  G^IOC$ALOALTMAPN
   ```

If IOC$ALOALTMAPN successfully allocates the alternate map registers, it stores the number of map registers allocated and the number of the first of the allocated map registers in CRB$L_INTD+VEC$W_NUMALT and CRB$L_INTD+VEC$W_MAPALT, respectively. It then returns with the low-order bit set in R0. Otherwise, it returns with the low-order bit of R0 clear.

5 If alternate map registers have been successfully allocated, set the alternate-map-lock bit in the CRB (VEC$V_ALTLOCK in CRB$L_INTD+VEC$W_MAPALT).

The driver-loading procedure calls the unit initialization routine once for each unit associated with the driver. If the unit initialization routine contains the code described previously, it permanently allocates a set of map registers for each CRB associated with the driver, which is a set of registers for each device controller that the driver serves. Because VMS records the allocation of standard and alternate map registers in separate areas of the ADP and CRB, a Q22 bus driver could permanently allocate a set of registers from both areas.

14.2.3 Loading Map Registers

Once a driver fork process has assigned a data path and allocated a set of map registers, it can request VMS to load the map registers with physical page-frame numbers (PFNs) by invoking the VMS macro LOADUBA.[3] LOADUBA calls the VMS routine IOC$LOADUBAMAP to load each allocated map register with the following data items:

- A bit setting to indicate whether the map register is valid.
- A bit setting to indicate whether the transfer is to start on the odd or even byte within a word; this bit is set if the low-order bit of UCB$W_BOFF is 1.
- The number of the data path to use for the transfer (UNIBUS drivers only).
- The page-frame number of a page in memory.
- A bit setting to indicate that the transfer operates in longword-aligned, random-access mode on the buffered data path; this bit is set when VEC$V_LWAE is set in VEC$B_DATAPATH (UNIBUS drivers only).

IOC$LOADUBAMAP loads the PFN of the first page of the transfer into the first allocated map register, the PFN number of the second page of the transfer into the second map register, and so forth. IOC$LOADUBAMAP sets the valid bit in every allocated map register except the last. It clears the valid bit in the final map register to prevent a prefetch from an invalid page.

[3] Q22 bus DMA device driver writers also use the LOADUBA macro to load a set of standard map registers (registers 0 through 495).

To calculate the PFN used in the I/O transfer, IOC$LOADUBAMAP uses three fields that VMS has written into the UCB:

- UCB$W_BOFF—Byte offset in the first page of the transfer
- UCB$W_BCNT—Number of bytes to transfer
- UCB$L_SVAPTE—Virtual address of the page-table entry that contains the PFN of the first page of the transfer

IOC$LOADUBAMAP determines the data path's number (for UNIBUS devices), the number of the first map register, the address of the first map register, and the number of allocated map registers from the CRB and the ADP, as follows:

UCB → CRB → number of the data path
UCB → CRB → number of first map register
UCB → CRB → ADP → virtual address of first map register
UCB → CRB → number of map registers

When IOC$LOADUBAMAP has loaded all the map registers and marked the last map register invalid, it returns control to the driver fork process.

VMS supplies a similar macro (LOADALT) and routine (IOC$LOADALTMAP) that Q22 bus drivers use to load a set of previously allocated alternate map registers (registers 496 through 8191). LOADALT and IOC$LOADALTMAP load the alternate map registers in the same manner as LOADUBA and IOC$LOADUBAMAP load standard map registers.

Drivers that handle UNIBUS byte-addressable devices call the routine IOC$LOADUBAMAPA. This routine performs the same function as IOC$LOADUBAMAP, with one exception. When IOC$LOADUBAMAPA loads map registers, it clears the byte-offset bit even if the transfer begins on an odd-byte address.

14.2.4 Computing the Starting Address of a Transfer

The driver fork process must calculate the starting address of a DMA transfer and load this address into the appropriate device register. UNIBUS drivers and other Q22 bus drivers that use a set of standard map registers take the following steps to make the calculation:

1. Write the byte-offset-in-page field of the UCB (UCB$W_BOFF) into bits 0 through 8 of a general register.
2. Get the number of the starting map register for the transfer from CRB$L_INTD+VEC$W_MAPREG. Write bits 0 through 6 of this 9-bit value into bits 9 through 15 of the general register.
3. Write bits 0 through 15 of the general register into the device's buffer address register.

4 Write bits 7 and 8 of the map register number, acquired in step 2, into the extended memory bits of the appropriate device register (usually the control and status register (CSR)).[4]

Q22 bus drivers that use a set of alternate map registers perform a similar procedure, as follows:

1 Write the byte-offset-in-page field of the UCB (UCB$W_BOFF) into bits 0 through 8 of a general register.

2 Get the number of the starting alternate map register for the transfer from CRB$L_INTD+VEC$W_MAPALT. Write bits 0 through 6 of this 13-bit value into bits 9 through 15 of the general register.

3 Write bits 0 through 15 of the general register into the device's buffer address register.

4 Write bits 7 through 12 of the map register number, acquired in step 2, into the extended memory bits of the appropriate device register (usually the control and status register (CSR)).

14.2.5 Computing the Transfer Length

Generally, a device driver must indicate to the device the size of a DMA transfer by writing to a device register. If a device expects the transfer size as a word count, for instance, the start-I/O routine computes the length of the transfer in words by dividing the byte count field of the UCB (UCB$W_BCNT) by 2. The routine loads the computed value into the device's word-count register. One of the FDT routines that processes the I/O request must ensure that the byte count for the transfer is even. An odd byte count results in the user's not receiving the last byte of data.

14.2.6 Activating the Device

Because a driver fork process can address device registers as though they were any other virtual address, the loading of the device buffer address register and CSR are simple procedures. The driver locates the CSR address of the device in the interrupt dispatch block (IDB), as follows:

UCB → CRB → IDB → CSR address

The CSR address is the virtual address of a device register. All other device registers are located at constant offsets from the CSR address. If, for example, the CSR is the first device register and the device's word-count register is the third device register, the device driver can describe the device register offsets and load the word-count register with the following series of instructions:

[4] One example of a device that does not treat the extended memory bits in this fashion is the DRV11–WA, the the driver code (XADRIVER) for which is listed in Appendix D. For the DRV11–WA, code in XADRIVER stores bits 7 and 8 of the map register number in a discrete device bus address extension register, then clears the extended address bits of the device's CSR. In contrast, XADRIVER handles the DR11–W according to the method described previously.

```
DEV_CSR =   0
DEV_XREG =  2
DEV_WDCNT = 4
.
.
.
; Compute word count of transfer and store it in user-defined UCB field,
; UCB$W_WDCNT.
.
.
.
      MOVL    UCB$L_CRB(R5),R4                ;Address of CRB
      MOVL    @CRB$L_INTD+VEC$L_IDB(R4),R4    ;Address of CSR
      MOVW    UCB$W_WDCNT,DEV_WDCNT(R4)       ;Move word count to device
                                              ;word count register
```

14.2.7 Completing a DMA Transfer

After a UNIBUS or Q22 bus driver fork process activates a DMA device, the driver waits for a device interrupt by invoking a VMS macro that suspends execution of the driver. When the device requests a hardware interrupt, the interrupt dispatcher gains control.

The dispatcher saves R0 through R5 and transfers control to the driver's interrupt service routine. If the interrupt service routine can match the interrupt with a suspended driver fork process, it reactivates the driver fork process at the point where execution was suspended. Most driver fork processes almost immediately invoke the VMS macro IOFORK.

IOFORK calls the VMS routine EXE$IOFORK. EXE$IOFORK saves the driver context (R3, R4, and PC) in the UCB fork block and inserts the address of the fork block (R5) in the processor-specific fork queue corresponding to the device's fork IPL. EXE$IOFORK then returns control to the driver's interrupt service routine, which dismisses the interrupt.

When the fork dispatcher reactivates the driver fork process, the driver performs any necessary cleanup operations, such as purging the data path and deallocating adapter resources used in the DMA transfer.

14.2.7.1 Purging the Data Path

Driver fork processes must purge the data path after the DMA transfer is complete. This is true for devices with buffered data paths, direct data paths, or no data path.

To purge the data path, the driver invokes the macro PURDPR, which in turn calls the VMS routine IOC$PURGDATAP. This routine takes the following steps to purge the data path:

1. Saves the contents of R4 on the stack.
2. Locates the CRB as follows:

   R5 → UCB → CRB

3. Obtains the starting address of the UNIBUS adapter's register space and stores it in R2.
4. Extracts the number of the data path to be purged from the CRB and loads it into R1.

5 Stores the address of the data path register in R4.

6 Instructs the UNIBUS adapter or Q22 bus interface to purge the data path. The routine then modifies R0 through R2 to contain the following information:

R0 Success/failure status. If the purge completes without error, the routine sets SS$_NORMAL in this register. If a data-path error does occur, R0 is clear and the hardware is reset.

R1 Contents of the data-path register.

R2 Address of the first adapter map register.

The address of the CRB remains in R3. This address, along with the information in R1 and R2, is used as input to the error logging routine in the event of a data-path error.

7 Restores the information stored on the stack to R4 and returns to the address in the driver immediately after the invocation of the PURDPR macro.

8 Some machine implementations also check for memory errors that might have occurred during the DMA operation, and, if an error is detected, log it.

If a data-path error occurs during a data-path purge, the driver should retry the entire DMA transfer.

14.2.7.2 Releasing a Buffered Data Path

A driver fork process releases a buffered data path by invoking the VMS macro RELDPR. RELDPR calls a VMS routine, IOC$RELDATAP, that determines which data path was assigned to the driver fork process and releases the data path to a waiting driver. The driver must be executing at fork IPL.

The data-path number is stored in the CRB. IOC$RELDATAP locates it as follows:

UCB → CRB → number of the data path

If the data path is permanently assigned to a device, IOC$RELDATAP does not release the data path. Otherwise, the data path number in the CRB (CRB$L_INTD+VEC$B_DATAPATH) is zeroed. The IOC$RELDATAP routine attempts to dequeue a waiting driver fork process from the data-path wait queue. It finds the queue as follows:

UCB → CRB → ADP → data-path wait queue

If another driver is waiting for a buffered data path, IOC$RELDATAP grants that driver fork process the data path, restores its context from its UCB fork block, and transfers control to the saved driver PC. When IOC$RELDATAP can allocate no more data paths, the routine returns to the driver that released the data path. This diversion of driver processing is transparent to the driver fork process.

If the data-path wait queue is empty, IOC$RELDATAP marks the data path as available in the ADP and returns control to the driver.

14.2.7.3 Releasing Map Registers

A driver fork process releases a set of map registers by invoking the VMS macro RELMPR at fork IPL. RELMPR calls the VMS routine IOC$RELMAPREG, which releases map registers in a manner similar to the way in which the RELDPR macro releases data paths. The CRB records the number of map registers assigned to the device. The number of the first map register and the number of map registers are located as follows:

UCB → CRB → number of the first map register
UCB → CRB → number of allocated map registers

IOC$RELMAPREG releases the map registers by adjusting the map-register-allocation information in the ADP.

Then, IOC$RELMAPREG attempts to dequeue a driver fork process from the standard-map-register wait queue. If a suspended driver is found, IOC$RELMAPREG takes the following steps:

1. Dequeues the fork block and restores driver context
2. Satisfies the map-register request, if possible
3. Reactivates the driver fork process at the instruction following the driver's request for map registers
4. Repeats steps 1 through 3

If the standard-map-register wait queue is empty or if IOC$RELMAPREG still does not have enough contiguous map registers for any of the waiting fork processes, it returns control to the fork process that released the map registers.

VMS supplies a similar macro (RELALT) and routine (IOC$RELALTMAP) that Q22 bus drivers use to release a set of alternate map registers (registers 496 through 8191). RELALT and IOC$RELALTMAP perform the release in the same manner as RELMPR and IOC$RELMAPREG. Note that VMS records the allocation of standard and alternate map registers in separate areas of the ADP and CRB.

14.3 Interrupt Dispatching in a UNIBUS/Q22 Bus System

The **interrupt dispatcher** is a combination of hardware and software that routes interrupts from a device on the UNIBUS or Q22 bus to the appropriate device driver's interrupt service routine. Although there are slight differences in the implementation of the interrupt dispatcher in different VAX systems, it performs the same tasks in any given VAX environment.

When a processor grants a device interrupt, the processor microcode first saves the PC and PSL of the currently executing code on the interrupt stack. The device responds to the grant by supplying a **device interrupt vector** in the range of 0 to 777_8 to the processor. VAX VMS uses the device interrupt vector to locate the correct interrupt transfer vector for the device. The interrupt transfer vector structure (VEC) contains a short

routine which issues a JSB instruction to the device driver's interrupt service routine. Execution continues at the location of the transfer vector.

This is a somewhat simplified view of the interrupt dispatcher's activities. Figure 14–5 and Figure 14–6 depict the flow of interrupt dispatching from the time that a processor grants the interrupt to the processing of the interrupt within a device driver's interrupt service routine. The following subsections provide a more complete description of the role of each component in the servicing of device interrupts.

Figure 14–5 Direct-Vector Interrupt Dispatching

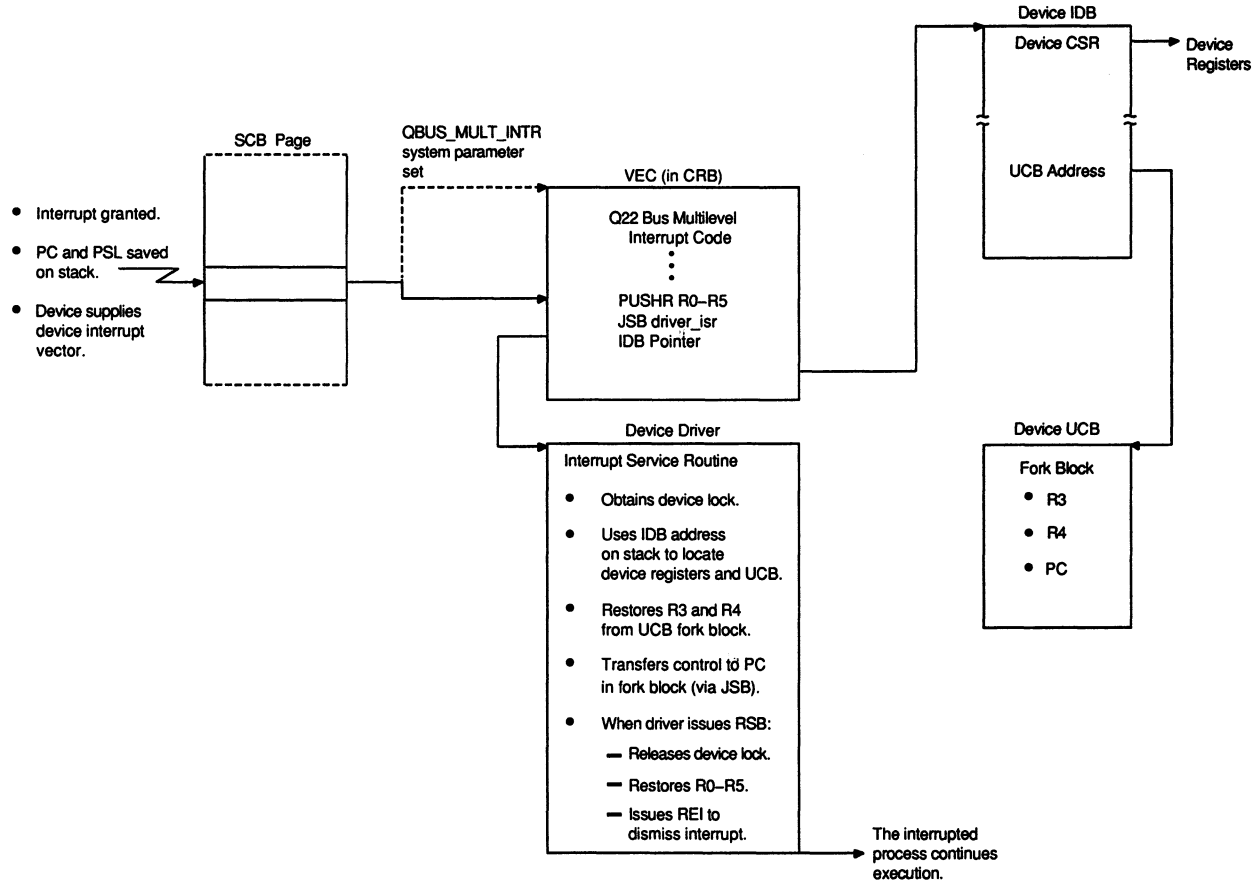
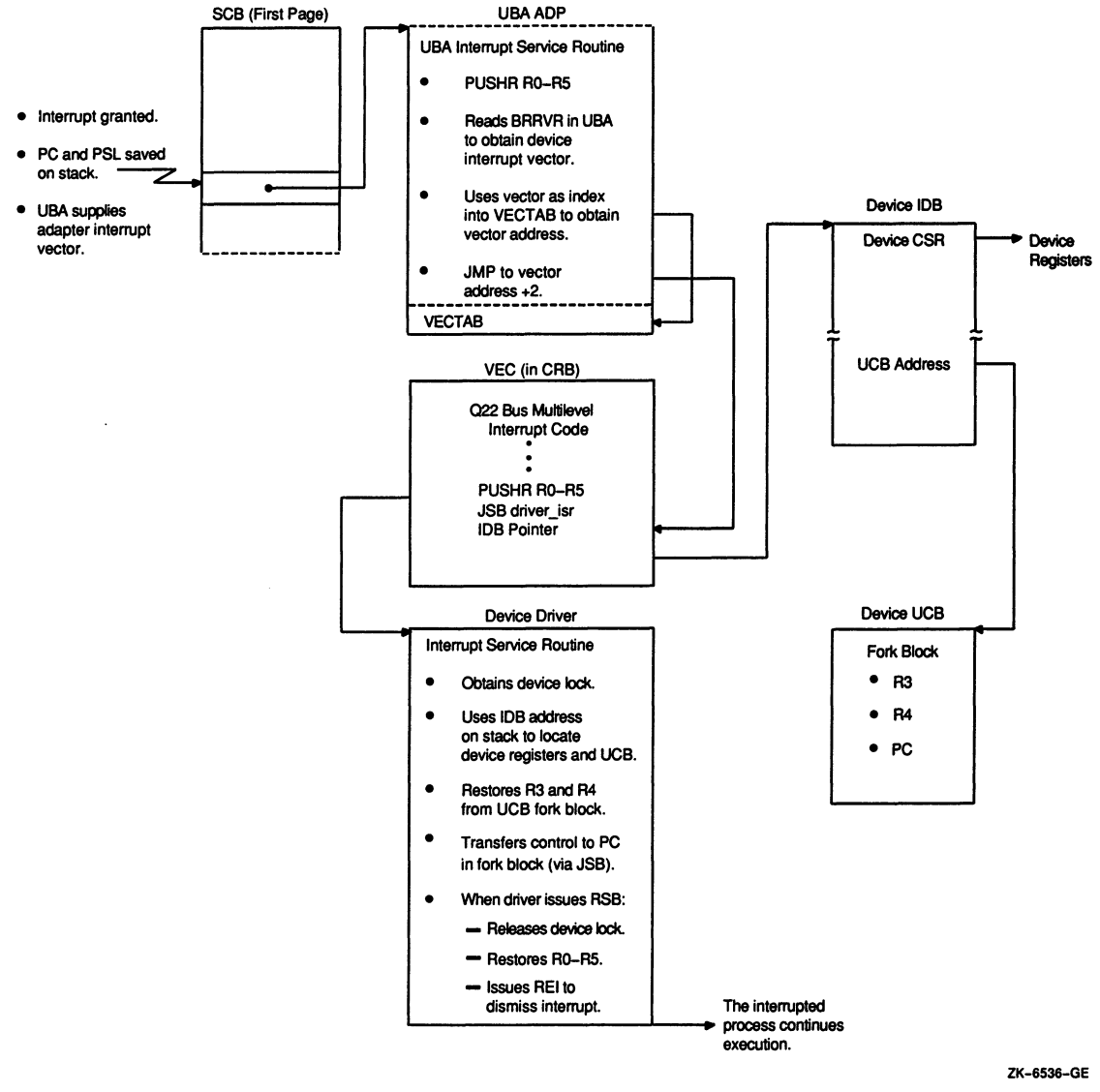


Figure 14–6 Non-Direct-Vector Interrupt Dispatching



14.3.1 Direct-Vector and Non-Direct-Vector Interrupt Dispatching

The system control block (SCB) contains the vectors that the VAX architecture uses to dispatch all interrupts and exceptions. The size of the SCB is system dependent. Page 1, the only SCB page defined by the VAX architecture, contains the addresses of software and hardware interrupt service routines and exception service routines.

The SCB therefore has an initial, albeit system-dependent, role in servicing device interrupts. A UNIBUS/Q22 bus system employs either of two methods to dispatch a device interrupt (see Table 14–2).

Table 14–2 VAX System UNIBUS/Q22 Bus Interrupt Dispatching

VAX System	Method of Interrupt Dispatching	Location of Adapter Dispatch Table
VAX–11/750, VAX–11/730, MicroVAX systems	Direct	SCB pages 2 and 3
VAX–11/780, VAX–11/785, VAX 8600, VAX 8650	Non-Direct	ADP vector-jump table
VAX 82*x*0/83*x*0	Direct	SCB page 2[1]
VAX 85*x*0/8700/88*x*0	Direct	SCB page 2[1]
VAX 6000 series	Direct	SCB page 2[1]

[1]Subsequent pages may be used if there is more than one DWBUA in the system.

- **Direct-vector interrupt dispatching:** If the system in question is a MicroVAX system, or uses a direct-vector UNIBUS adapter (UBA), it dispatches a device interrupt directly through page 2 (or subsequent pages, for VAX systems with more than one such UNIBUS) of the SCB. It takes the device interrupt vector and uses it as an index into the appropriate SCB page, thus obtaining the address of the appropriate interrupt transfer vector structure (VEC) for the device.
- **Non-direct-vector interrupt dispatching:** In a VAX system that employs a non-direct-vector UNIBUS adapter, the adapter posts an interrupt that is dispatched through a vector in page 1 of the SCB that points to a UBA interrupt service routine. For each non-direct-vector UBA adapter, the VMS adapter initialization procedure creates four such interrupt service routines, each corresponding to a device BR (bus request) level, and places them in an area of nonpaged pool specially allocated at the end of the adapter control block.

  The UNIBUS adapter's interrupt service routine performs the following actions:

  1 Saves R0 through R5 on the interrupt stack.

  2 Reads the UNIBUS adapter's Bus Request Receive Vector register (BRRVR) to determine the vector address of the device requesting the interrupt.

  3 Uses the vector address as an index into the adapter dispatch table to locate the interrupt transfer vector for the device in the CRB. For each non-direct-vector UBA, an adapter dispatch table (also known as the vector-jump table) is located after the UBA interrupt service routines in nonpaged pool.

  4 Transfers control by means of a JMP instruction to a location within the interrupt transfer routine contained in the VEC structure.

From another point of view, direct-vector interrupt dispatching and non-direct-vector interrupt dispatching are characterized by the following two factors:

- Location of the adapter dispatch table
- Contents of the interrupt transfer routine

The following sections further examine the differences in the dispatching methods. Figures 14–5 and 14–6 present the flow of tasks performed within the context of both direct-vector and non-direct-vector interrupt dispatching.

14.3.2 Adapter Dispatch Table

The **adapter dispatch table** contains 128 longword vectors, each of which corresponds to a device interrupt vector. Each longword vector within the adapter dispatch table contains either the address of an **interrupt transfer vector** structure (VEC), located within the channel request block (CRB) of the device's controller or, if no device is using the vector, the address of the adapter's unexpected interrupt service routine. In either case, the address contained in the adapter dispatch table is longword aligned.

The location of the adapter dispatch table, as signified by the contents of ADP$L_VECTOR, is system dependent:

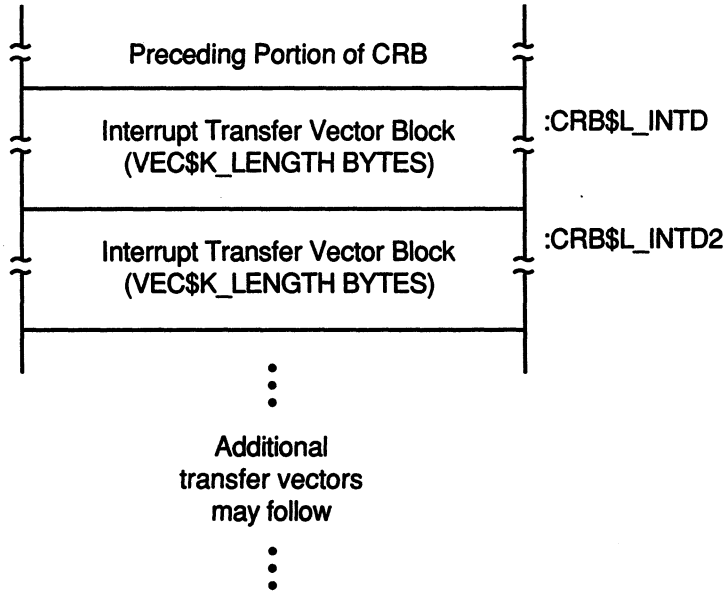
- MicroVAX systems and those VAX systems that employ direct vector UNIBUS adapters situate the adapter dispatch table in the second and subsequent pages of the system control block (SCB), as described previously.
- Those VAX systems that employ non-direct-vector interrupt dispatching situate the adapter dispatch table in a region of nonpaged pool (known also as the vector-jump table and commonly referred to as VECTAB).

14.3.3 Interrupt Transfer Vector and Interrupt Transfer Routine

The **interrupt transfer vector** data structure (VEC) is located within the channel request block (CRB) corresponding to the interrupting device's controller, as shown in Figure 14–7.

The interrupt transfer vector structure (see Figure 14–8 and the CRB structure in the *VMS Device Support Reference Manual*) starts with several lines of executable code known as the interrupt transfer routine. It also stores several pieces of data, including pointers to the unit and controller initialization routines in the device driver, the address of the interrupt dispatch block (IDB), and the address of the adapter control block (ADP). The interrupt transfer vector may also include information reflecting the disposition of the adapter's map registers.

Figure 14–7 VEC Structures Within a CRB



There may be one or more interrupt transfer vectors within a single CRB, as shown in Figure 14–7. VMS creates the appropriate number of interrupt transfer vector structures within a CRB according the the value specified in the /NUMVEC qualifier to the SYSGEN command CONNECT. The default value is 1.

VMS automatically initializes the interrupt dispatching instructions and the data structure locations in each of the specified vectors.

The **interrupt transfer routine** is a piece of executable code at the beginning of each interrupt transfer vector. It is the interrupt transfer routine that ultimately transfers control to the device driver's interrupt service routine and, to a certain extent, establishes the context for its execution.

For those VAX systems employing non-direct-vector interrupt dispatching, the interrupt transfer routine consists of only one instruction:

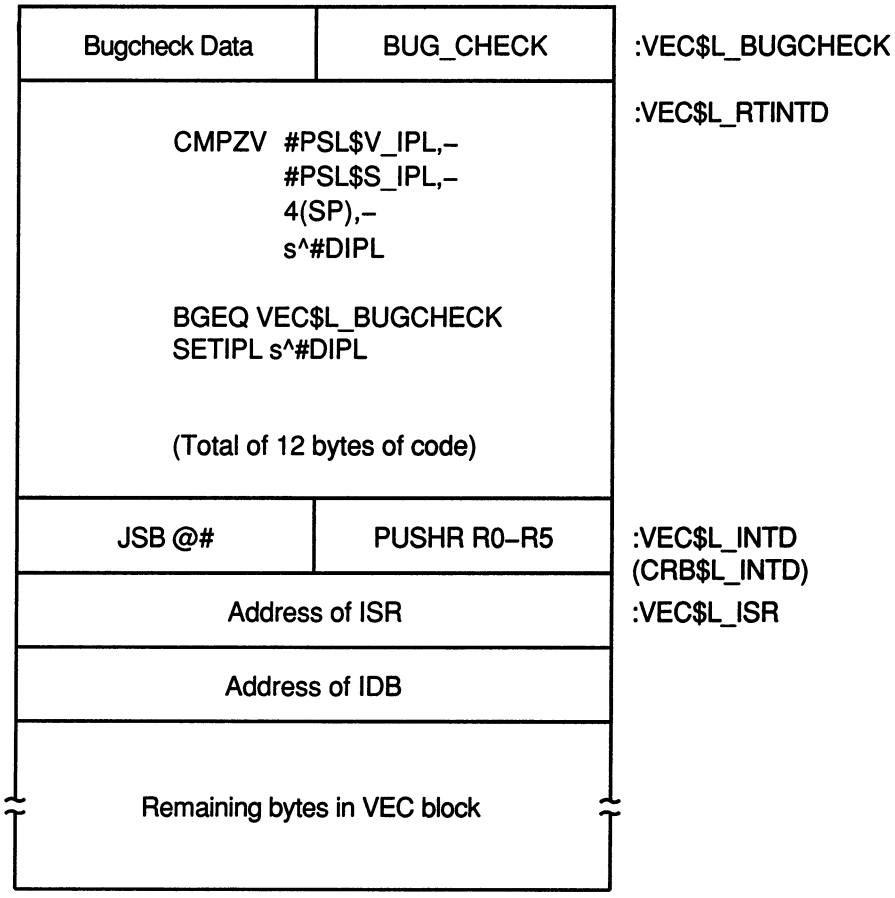
```
JSB     @#^driver-isr-address
```

For those VAX systems employing direct-vector interrupt dispatching, the interrupt transfer routine consists of the following two instructions:

```
PUSHR   #^M<R0,R1,R2,R3,R4,R5>
JSB     @#^driver-isr-address
```

Note: If the MicroVAX system has multilevel device interrupt dispatching enabled, these two instructions are preceded by some instructions that check the legality of the Q22 bus configuration and conditionally lower IPL. See Section 14.3.4 for a description of this optional function of the interrupt transfer routine.

Figure 14–8 Interrupt Transfer Vector Block (VEC)



The driver-loading procedure obtains the address of the interrupt service routine for each interrupt transfer vector structure from the reinitialization portion of the driver prologue table (see Section 6.1). This section of the DPT contains one or more DPT_STORE macros that identify the addresses of the interrupt service routines. For example:

```
DPT_STORE,CRB,CRB$L_INTD+VEC$L_ISR,D,isr_for_1st_vector
DPT_STORE,CRB,CRB$L_INTD2+VEC$L_ISR,D,isr_for_2nd_vector
DPT_STORE,CRB,CRB$L_INTD+<2*VEC$K_LENGTH>+VEC$L_ISR,D,isr_for_3rd_vector
```

The number of DPT_STORE macros that identify interrupt service routines must equal the number of vectors given in the /NUMVEC qualifier to the SYSGEN command CONNECT to avoid errors in device initialization or interrupt handling.

Immediately following the interrupt transfer routine in the CRB is the address of the interrupt dispatch block (IDB) associated with the CRB. When the JSB instruction executes, a pointer to the address of the IDB is pushed onto the top of the stack as though it were a return address. The driver interrupt service routine can use this IDB address as a pointer

into the I/O database. See Figures 14–5 and 14–6 for an illustration of the context available to a driver's interrupt service routine when it is called by the interrupt transfer routine.

14.3.4 Multilevel Device Interrupt Dispatching for Q22 Bus Devices

VAX peripheral devices request interrupts at IPLs 20 through 23. IPLs 20 through 23 generally correspond with the four bus request levels of the UNIBUS (BR4 through BR7) and Q22 bus (BIRQ4 through BIRQ7).

The UNIBUS also has four bus grant lines (BG4 through BG7). Because of this, interrupt dispatching for UNIBUS devices inherently occurs at four levels. When a UNIBUS device requests an interrupt at BR4, for example, from a processor executing at an IPL lower than IPL 20, the processor grants the interrupt to the device at IPL 20 (BG4). If the processor is already executing at IPL 20 or above, the device interrupt remains pending.

The MicroVAX Q22 bus implementation has but one bus grant line (BIAK). As a result, the central processor must, by default, grant all Q22 bus device interrupts at a single IPL (IPL 23), even though it arbitrates interrupt requests according to the bus request line used. When a Q22 bus device requests an interrupt at BIRQ4, for example, from a processor executing at an IPL lower than IPL 20, the processor grants the interrupt, unconditionally raising IPL to IPL 23. If the processor is already executing at IPL 20 or above, the interrupt remains pending.

There are certain consequences of this implementation of interrupt dispatching on the configuration and behavior of Q22 bus devices:

1. Because MicroVAX systems dispatch Q22 bus interrupts at a single IPL, it is essential that Q22 bus devices that request interrupts at a high BIRQ be positioned on the bus closer to the CPU than devices that interrupt at a low BIRQ. (To determine the BIRQ level of any given Q22 bus device, refer to its hardware user's guide.)
2. It is possible for a Q22 bus peripheral that requests interrupts at a low BIRQ to block the granting of an interrupt to a peripheral that requests interrupts at a higher BIRQ. For instance, the processor could grant an interrupt to a BIRQ4 device, elevating its IPL to IPL 23 in the process. While executing at IPL 23, the processor would not grant the interrupt request of a BIRQ7 device. In a real-time environment, where I/O operations to one peripheral must always have priority over lesser forms of I/O, this behavior can cause problems.

VMS incorporates a means by which system programmers, concerned about real-time performance issues, can avoid these problems and implement multilevel interrupt dispatching for devices on a MicroVAX Q22 bus.

Because single-level interrupt dispatching is sufficient for most applications, only system managers and system programmers involved in real-time system environments should attempt to use the multilevel device interrupt dispatching capability of VMS Version 5.4. Such users should possess a thorough understanding of the VMS interrupt

dispatching mechanism and the means by which VMS synchronizes access to structures in the I/O database.

If you must enforce real-time device priorities in your Q22 bus system, you can do so by setting the QBUS_MULT_INTR system parameter. This static parameter causes VMS to set up the proper code and data structures to enable multilevel device interrupt dispatching at system initialization.[5]

When you bootstrap a MicroVAX system with the QBUS_MULT_INTR system parameter set, VMS initializes data structures for each device and implements multilevel device interrupts as follows:

- Locates the address of the device driver's interrupt service routine in the driver's DPT and stores it in the appropriate VEC data structure.
- Adjusts the corresponding vectors in the second page of the SCB so that they point to the multilevel device interrupt dispatching code in the interrupt transfer vector (that is, replacing the address of CRB$L_INTD with CRB$L_INTD+VEC$L_RTINTD).
- Sets up data and code at VEC$L_RTINTD in the first interrupt transfer vector for the device. This code performs special checks for the legality of the Q22 bus configuration and conditionally lowers IPL, when necessary, to service the interrupts of low priority devices.

When multilevel device interrupt dispatching is enabled, interrupt dispatching proceeds as in the default case. However, after the processor raises IPL to IPL 23 to grant the interrupt, the dispatching of the interrupt results in the lowering of IPL to device IPL, if necessary, to service the interrupt. As a result, the processor, executing at the lower IPL, will be free to grant interrupts from higher priority devices.

Prior to activating this feature in a MicroVAX system, a user should perform the following tasks:

1. Ensure that the Q22 bus is properly configured.
2. Adapt any existing non-Digital-supplied device driver so that it correctly initializes any secondary vector (VEC) structures it uses and also refers to fields in the channel request block (CRB) and VEC by the proper symbolic offsets.

14.3.4.1 Ensuring That the Q22 Bus Is Properly Configured

The MicroVAX Q22 bus architecture mandates that devices with the ability to interrupt at a high BIRQ level (for instance, BIRQ7) must be positioned on the bus closer to the CPU than devices that interrupt at lower BIRQ levels. In a Q22 bus system, when the processor grants an interrupt to a device, the processor passes the interrupt down the BIAK line to the first device on the bus. If the device is not the one requesting an interrupt, it is responsible for propagating the acknowledgment grant to the next device on the bus, and so on. However, if a device coincidentally initiates an interrupt just before it would be required to pass the grant further down the bus, it may instead "steal" the grant for itself.

[5] Other VAX systems ignore the QBUS_MULT_INTR system parameter.

The problems of an illegally configured Q22 bus can be illustrated in the following two examples.

Consider the case of a device that interrupts at BIRQ6 (IPL 22). Assume the CPU is at IPL 21. The bus arbitrator will grant the device its interrupt and send the grant down the Q22 bus. However, if at this precise moment a device that is closer to the CPU on the grant path, and that interrupts at BIRQ5 (IPL 21), initiates an interrupt, it may not propagate the grant as it should. Instead, it may steal the grant and assume ownership of the interrupt. Thus, there exists the possibility of an IPL 21 device successfully interrupting a processor executing at IPL 21. The end result is an unpredictable break in synchronization that could have a multitude of consequences.

Consider also an instance where the second device interrupts at a lower IPL, such as BIRQ4 (IPL 20). Although it essentially is executing below the IPL of the CPU (IPL 21), this device, too, may steal the interrupt grant. The break in synchronization in this instance will result in a reserved operand fault when the driver's interrupt service routine issues the REI instruction, as the VAX architecture does not permit an REI from a lower IPL to a higher IPL.

If the MicroVAX system employs the multilevel device interrupt dispatching option, VMS introduces special code in the interrupt transfer vector data structure that helps prevent violations of system synchronization resulting from an illegally configured Q22 bus (see Figure 14–8). This code, located at offset VEC$L_RTINTD, checks the device IPL of the interrupting peripheral against the IPL in the processor status longword (PSL) of the interrupted thread of code. If the device IPL is not greater than the IPL in the saved PSL, VMS generates an ILLQBUSCFG bugcheck, signifying that the Q22 bus is illegally configured.

14.3.4.2 Effects of Enabling Multilevel Device Interrupt Dispatching

Before enabling multilevel device interrupts in a MicroVAX system, you should first ensure that all existing device drivers have been adapted according to the following guidelines:

- If the driver creates or accesses any secondary VEC data structures, it must take steps to initialize properly the multilevel device interrupt dispatching code (at offset VEC$L_RTINTD in the primary VEC structure) in these secondary structures. Failure to properly initialize these structures in the system can negate any performance increases expected as a result of enabling multilevel device interrupt dispatching.
- The device IPLs of certain Q22 bus devices may differ from those of corresponding UNIBUS devices. To ensure that the DPT specifies the correct device IPL, refer to the device's hardware user's guide.
- Wherever the driver implicitly refers to the longword in the VEC structure that contains the address of the interrupt service routine, it should explicitly use the symbol VEC$L_ISR. For example, you should replace any instance of CRB$L_INTD+4 with CRB$L_INTD+VEC$L_ISR.

- If the driver assumes that the contents of the device's SCB vector (that is, the vector in the adapter dispatch table) always points to CRB$L_INTD, it must be appropriately modified to reflect the implementation of multilevel device interrupt dispatching. The transfer vector can legally point to any longword-aligned address between CRB$L_INTD+VEC$L_RTINTD and CRB$L_INTD+VEC$L_INTD. Because MicroVAX systems utilize direct-vector interrupt dispatching, the transfer vector will never point to VEC$L_INTD+2.

Although certain of the symbolic offsets defined in the data structure definition macro $VECDEF have negative values, driver code can uniformly refer to the contents of the VEC structure in the form CRB$L_INTD+VEC$*x_symbol*.

15 MASSBUS Device Support

The MASSBUS adapter (MBA) is the hardware interface between the backplane interconnect and MASSBUS storage devices. The MASSBUS is the communication path linking the MASSBUS adapter to the mass storage devices.

The MASSBUS adapter performs the following functions that allow communication between devices and memory:

- Mapping of virtual addresses to physical addresses
- Buffering of data for transfers between main memory and the MASSBUS
- Transfer of interrupts from MASSBUS devices to the backplane interconnect

A MASSBUS adapter supports any combination of up to eight device controllers. Typical MASSBUS controllers include the TM03 tape controller and the RP06, RM03, and RM80 disk controllers. Only one controller can transfer data over the MASSBUS at a time.

The TM03 tape controller supports up to eight tape drives. In contrast to tape controllers, there is a one-to-one relationship between a disk controller and its device; each controller supports only one disk drive. The VMS system interprets and maintains the I/O database differently, depending upon whether the controller is single unit or multiunit.

Each MASSBUS controller connected to a MASSBUS adapter is assigned a unit number in the range 0 to 7. The method of unit number assignment is controller specific, but you can obtain the number from either unit plugs or switch packs. In the case of a controller for several devices, the unit number is distinct from the subunit numbers assigned to the individual drives connected to the controller.

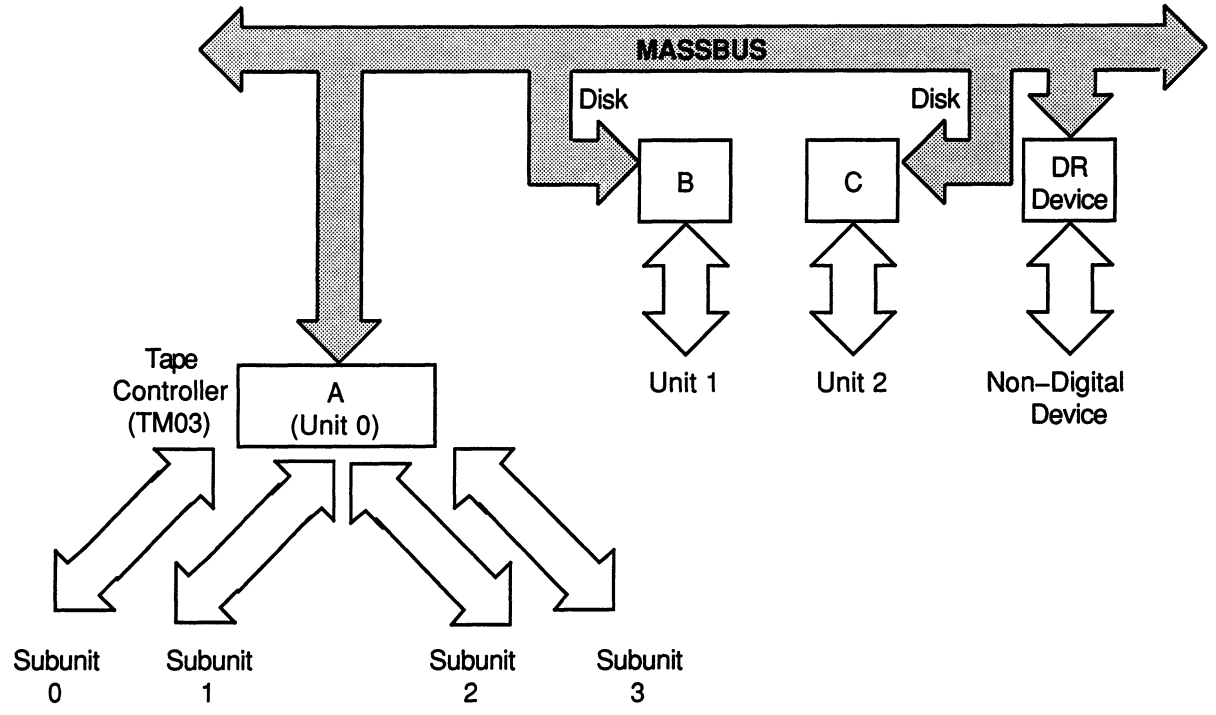
Figure 15–1 illustrates a possible MASSBUS configuration.

15.1 MASSBUS Adapter Registers

The MASSBUS adapter has three sets of registers:

- The MASSBUS adapter's registers
- External registers for each device (controller) on the MASSBUS
- 256 map registers

Figure 15–1 MASSBUS Configuration



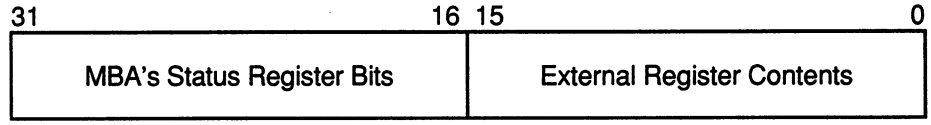
To allow competing devices to share these resources, access to and modification of all MASSBUS adapter registers (internal, external, and map registers) are governed by certain rules and conventions. In particular, access to registers might, at times, require ownership of either the device controller or the MASSBUS adapter itself, or both. Subsequent sections in this chapter discuss the methods of obtaining such ownership of these shared resources.

MASSBUS adapter external registers are device dependent and accessible whether or not the driver owns the MASSBUS adapter. However, in the case of multiunit MASSBUS adapter controllers, the driver might need to own the controller before it can gain access to a register.

MASSBUS adapter external registers are each 16 bits wide, but they must be accessed as longwords. When a driver reads an external register, the MASSBUS adapter concatenates the high-order 16 bits of the MBA's status register (one of the MBA's internal registers) with the contents of the specified external register. Figure 15–2 illustrates the resulting longword.

On a write to an external register, the MASSBUS adapter uses the low-order 16 bits of the longword source operand to update the external register.

Figure 15–2 MASSBUS External-Register Longword



ZK–1796–GE

MASSBUS adapter internal and map registers are 32 bits in length. They must be accessed as longwords or the processor will signal a machine check exception. The driver for a MASSBUS device must obtain exclusive ownership of the MASSBUS adapter before modifying any of the MBA's internal or map registers.

Bits 21 through 30 of each of the MBA's map registers are reserved; they cannot be written. Use of the MBA's map registers is analogous to use of the UNIBUS adapter's map registers with the following exceptions:

- Because the MASSBUS can handle only one transfer at a time, ownership of the MASSBUS adapter implies ownership of all its map registers. Thus, the driver need not independently request map registers.
- The MBA's map registers do not contain a byte-offset field. The driver loads the full MASSBUS adapter virtual address, including the byte alignment, into the MASSBUS adapter virtual address register (MBA$L_VAR, one of the MBA's internal registers) at the start of a data transfer. Use of the MBA$L_VAR register is described in Section 15.1.1.
- The MBA's map registers do not contain a data path field; the MASSBUS adapter has a single data path, and ownership of the adapter implies ownership of the path. Thus, the driver need not allocate the data path independently.

15.1.1 Loading MASSBUS Adapter Registers

To prepare for a data transfer over the MASSBUS, the driver that owns the MASSBUS adapter uses the LOADMBA macro to load the MBA's map registers and associated internal registers. The LOADMBA macro invokes the subroutine IOC$LOADMBAMAP, which performs the following steps:

- Determines the number of map registers needed to map the data area by adding the contents of UCB$W_BCNT to UCB$W_BOFF, adjusting the sum to the next even multiple of 512, and dividing the result by 512.
- Loads the specified number of map registers, beginning with map register 0, with the contents of the page-table entries to which UCB$L_SVAPTE points. This step maps the data area for the transfer into the low portion of the MBA's virtual address space. The routine also loads the next map register beyond the number used to map the

data area with zeros (an invalid map entry). This procedure stops the transfer should a hardware failure occur.

- Loads the MBA$L_VAR register with the zero-extended contents of UCB$W_BOFF. Because the first byte of the data area is located at offset UCB$W_BOFF within the page of memory mapped by map register 0, the UCB$W_BOFF contains the virtual address of the start of the data area in MASSBUS adapter virtual address space.
- Loads the complement (negative) of UCB$W_BCNT into the MBA's byte-count register (MBA$L_BCR).

Note that if a driver is to perform a data transfer in the reverse direction (for example, read reverse on a tape), it must modify the contents of the MBA$L_VAR, as established by IOC$LOADMBAMAP, so that it points to the last byte of the data area. This is done by adding one less than the contents of UCB$W_BCNT to the contents of the MBA$L_VAR register.

During the progress of a data transfer over the MASSBUS, the MBA$L_VAR register is continuously updated so that it points to the current position in the data area. The *VAX Hardware Handbook* illustrates the mapping of the contents of the MBA$L_VAR register into physical memory.

15.1.2 MASSBUS Adapter Registers and Offsets

During system initialization, VMS builds an adapter control block (ADP), a channel request block (CRB), and an interrupt dispatch block (IDB) for each MASSBUS adapter. The system also allocates 4KB of system virtual address space for the adapter's register I/O space. The base of this I/O register virtual address space is placed in IDB$L_CSR. Thus, you can access MASSBUS adapter registers using the base register virtual address plus some offset. The $MBADEF macro defines the offsets for MASSBUS adapter registers. The major symbols defined by this macro are shown in Table 15–1.

Table 15–1 Major Offsets Defined by $MBADEF

Symbol	MBA Register Name	Hex Offset
MBA$L_CSR	Configuration register	0
MBA$L_CR	Control register	4
MBA$L_SR	Status register	8
MBA$L_VAR	Virtual-address register	C
MBA$L_BCR	Byte-count register	10
MBA$L_DR	Diagnostic register	14
MBA$L_SMR	Selected map register	18

(continued on next page)

Table 15–1 (Cont.) Major Offsets Defined by $MBADEF

Symbol	MBA Register Name	Hex Offset
MBA$L_CAR	Command-address register	1C
MBA$L_ERB	External register base	400
MBA$L_AS	Attention-summary register	414
MBA$L_MAP	Base of map registers	800

The MASSBUS adapter's internal registers occupy the low-order 1024 bytes of address space even though there are only eight internal MBA registers. Beyond the internal registers, there are eight blocks of 32 longwords (128 bytes) each, one block for each of the eight device controllers that can be connected to a single MASSBUS adapter. Each of these blocks provides space for the device registers of each controller. Beyond the device-register space is the area reserved for the MASSBUS adapter's 256 map registers.

Figure 15–3 illustrates the relative positions of the MASSBUS adapter's registers and the values device drivers use to gain access to them. The base address of the MASSBUS adapter's address space, stored in IDB$L_CSR, is the address of the first of the MASSBUS adapter's internal registers.

IDB$L_CSR represents the internal register's virtual location, while the MBA$L_ symbols represent register values as defined by $MBADEF. Note that the MASSBUS adapter's register space occupies only the first 3Kb out of the 8Kb allotted to physical I/O address space. However, by convention, VMS allocates 4Kb of virtual addresses to each MASSBUS adapter.

To address a map register in the MASSBUS adapter, the driver constructs the following address:

IDB$L_CSR + MBA$L_MAP + map-register-index

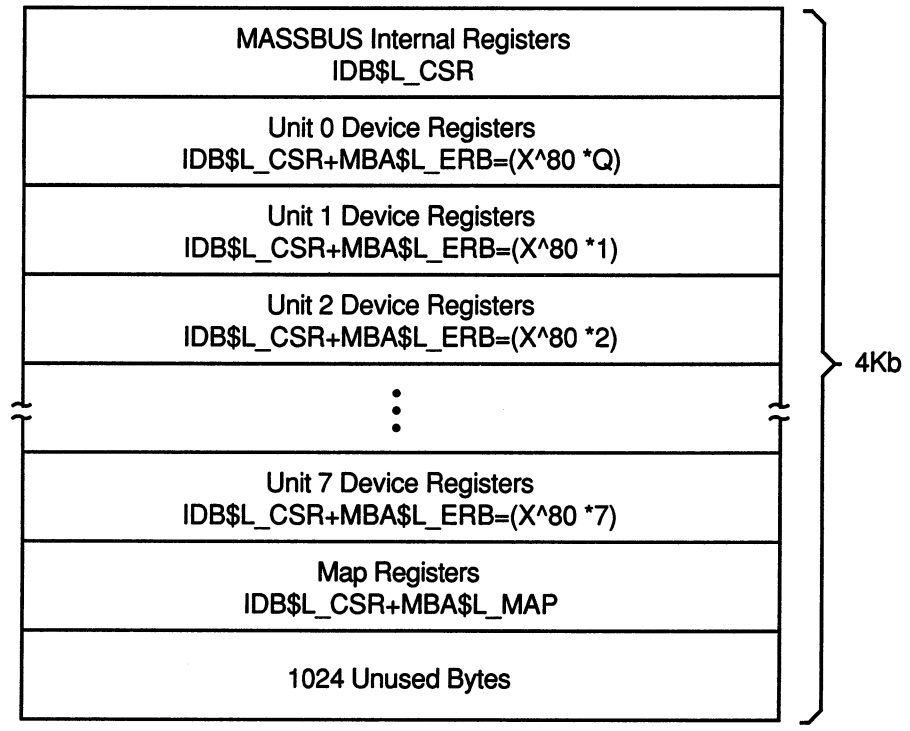
To address a device register, the driver constructs the following address:

IDB$L_CSR + MBA$L_ERB + (unit-number * 80_{16}) + register-displacement

An individual driver should define offsets for the registers of its device. During execution, the driver computes a register address by summing the MBA's starting virtual address (the contents of IDB$L_CSR), MBA$L_ERB, the unit number of the device controller multiplied by 80_{16}, and the offset of the specified register.

The attention-summary register (MBA$L_AS), as shown in Table 15–1, appears to reside within the external-register space reserved for MASSBUS adapter controller 0. Actually, the attention-summary register is a composite register. Each of the MASSBUS adapter's controllers contributes one bit of information to the register. This composite register appears in each of the eight device register spaces at offset 10_{16} from the base of the device registers for that device. Thus, MBA$L_AS can be

Figure 15–3 Location of MASSBUS Registers in Physical Address Space



defined as any of the values 410_{16}, 490_{16}, 510_{16}, 590_{16}, and so on. For convenience, it has been defined as 410_{16}.

15.1.3 Modifying MASSBUS Adapter Registers

The driver for a MASSBUS device must obtain ownership of the MBA before modifying any of the MBA's internal registers or map registers. A driver obtains ownership of the MBA by invoking either the REQPCHAN macro or the REQSCHAN macro, depending on whether the device is connected to a single-unit MASSBUS controller or a multiunit MASSBUS controller.

For dedicated controllers, invoke the REQPCHAN macro. Because the controller is dedicated to its single device, there is never any contention for the controller.

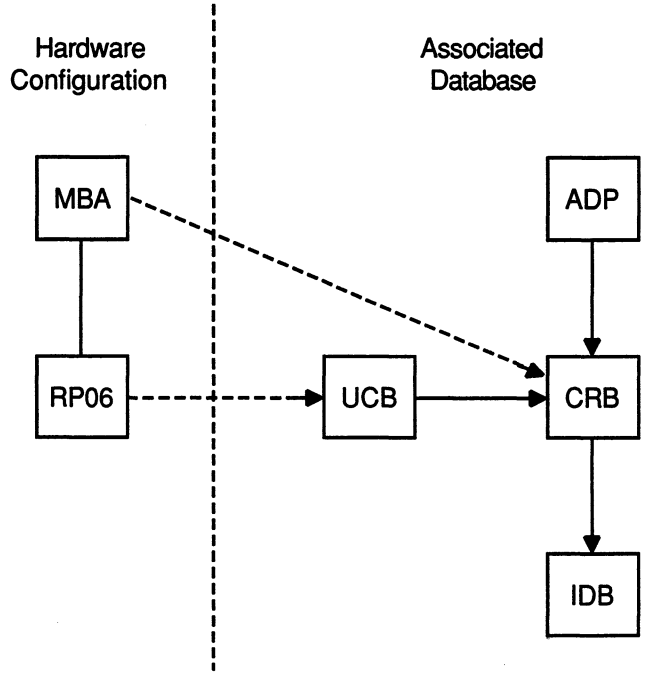
For multiunit devices, however, invoke the REQSCHAN macro to obtain MBA ownership because several devices can share the controller, and so must contend for its use. The controller for several devices relegates the MASSBUS adapter to a secondary position. Thus, for multiunit controllers, invoke REQPCHAN to gain ownership of the controller, and invoke REQSCHAN to obtain the MASSBUS adapter.

15.2 I/O Database for MASSBUS Devices

During initialization, the system creates an ADP, a CRB, and an IDB for each MASSBUS adapter included in the configuration. The driver-loading procedure subsequently builds additional data structures for each device controller connected to a MASSBUS adapter. The type of structure created depends upon whether the device controller is a dedicated controller or the controller of several devices.

The system builds a unit control block (UCB) for each single-unit controller. Figure 15–4 illustrates the I/O database for a MASSBUS adapter with one dedicated controller attached to it. Note that the ADP, CRB, and IDB all correspond to the MASSBUS adapter and can logically be considered a single, extended data block. The UCB corresponds to the device/controller pair. Because of the one-to-one correspondence between a dedicated controller and its device, the system does not need to distinguish between the two and thus does not maintain separate data blocks for each piece of hardware.

Figure 15–4 I/O Database for MASSBUS Disk Unit



A controller of several devices, however, requires separate data structures for the controller and each of its subunits (devices). The driver-loading procedure builds a CRB/IDB pair for the controller, as well as a UCB for each subunit. Figure 15–5 shows the I/O database created for a MASSBUS adapter with one disk unit and two tape units.

Figure 15–5 I/O Database for MASSBUS Disk and Tape Units

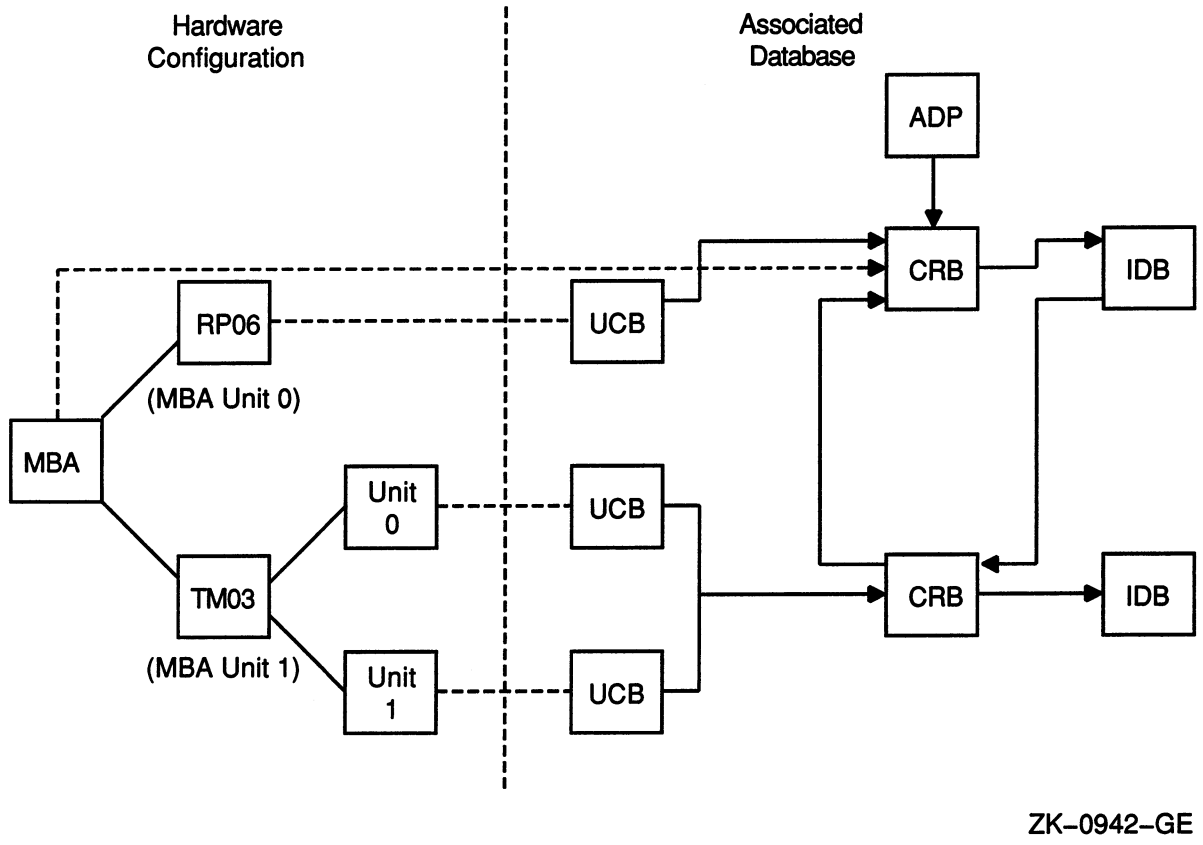
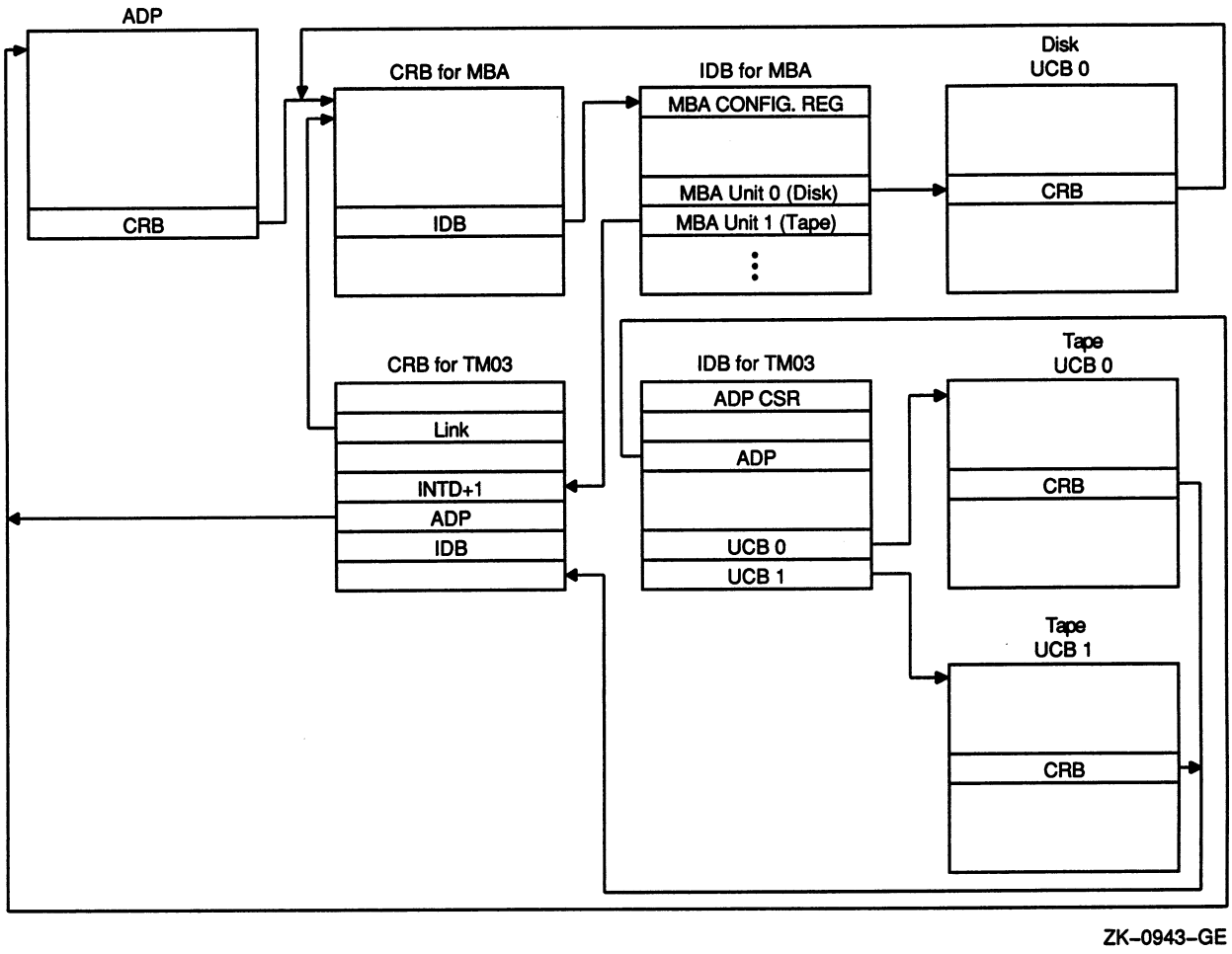


Figure 15–5 does not include several pointers used in interrupt dispatching. In particular, the IDB associated with the MASSBUS adapter maintains an array of up to eight longwords that point to the data structures associated with the eight possible MASSBUS controllers attached to the MASSBUS.

For dedicated controllers, the IDB longword points to the device's UCB; whereas, for a controller for several devices, the longword (or longwords) points to a field within the CRB associated with the controller. The low bit of this longword, when set, indicates a multiunit vector. The software checks this bit to determine whether the longword points to a single UCB or a multiunit CRB.

Also not pictured in Figure 15–5 is how multiunit IDBs also maintain an array of longwords. Each longword points to the individual UCBs for the units attached to the controller. Figure 15–6 illustrates in more detail the set of I/O data structures for the MASSBUS adapter and its devices.

Figure 15–6 I/O Data Structures Used in Dispatching a MASSBUS Device Interrupt



15.3 MASSBUS Adapter Operations

The MASSBUS accepts two kinds of operations: data transfer operations and nondata transfer operations. Data transfer operations require the use of MASSBUS adapter shared resources, while nondata transfers do not.

Before a driver can activate a data transfer operation on the MASSBUS, the driver must request and receive ownership of the MASSBUS adapter on behalf of the device unit. However, drivers must not initiate nondata transfer operations while they have control of the MASSBUS adapter. Section 15.4.1 explains this statement further.

The MASSBUS adapter generates interrupts when data transfers terminate and when attention conditions arise on devices. When an interrupt occurs on the MASSBUS adapter, the MASSBUS adapter's interrupt dispatcher determines whether the interrupt is for a data transfer or an attention condition.

Data transfer interrupts occur when a data transfer either completes or is aborted. When the interrupt occurs, the MBA's status register (MBA$L_SR) contains information about the condition that caused the interrupt.

Attention interrupts occur when nondata transfers on MASSBUS devices terminate, or when the device undergoes an exceptional condition, such as coming on line.

The MASSBUS adapter's attention-summary register controls attention-interrupt handling. This register contains eight bits of data, one for each of the eight possible controllers that can be connected to the MASSBUS adapter. When a device incurs an attention condition, the hardware sets the corresponding bit in the attention-summary register and generates a MASSBUS adapter interrupt.

If the attention condition occurs while a data transfer operation for another device is in progress, the hardware sets the bit in the attention-summary register but suppresses the attention interrupt. The interrupt generated when the data transfer is completed allows the MASSBUS adapter's interrupt dispatcher to gain control, handle the data transfer interrupt, check the attention-summary register and then invoke the proper driver to handle the attention condition.

15.4 MASSBUS Adapter's Interrupt Dispatching

When interrupts occur on the MASSBUS adapter, the MASSBUS adapter's interrupt dispatcher gains control. This routine first determines whether the interrupt is the result of a data transfer or an attention condition. The routine checks to see if the MASSBUS adapter is owned and, if so, by whom.

15.4.1 Checking for MASSBUS Adapter Ownership

There are two conditions by which the interrupt dispatcher can determine that the interrupt is an attention interrupt:

- If the MASSBUS adapter is not owned
- If the MASSBUS adapter is owned, but the owner is not expecting an interrupt (UCB$V_INT in UCB$L_STS is clear)

When the MASSBUS adapter is owned and the owner expects an interrupt, the interrupt is assumed to be the result of a data transfer operation.

As mentioned earlier, a driver must not initiate nondata transfers on the MASSBUS adapter while it owns the adapter. For example, consider a MASSBUS adapter attached to two disk units, A and B. Disk A is performing an IO$_SEEK (a nondata transfer operation that completes fairly quickly), while at the same time, disk B is performing an IO$_RECAL operation (a nondata transfer operation that takes about 0.5 seconds to complete).

The driver for disk A correctly initiates its operation without obtaining possession of the MASSBUS adapter channel, but the disk B driver initiates its operation while it owns the MASSBUS adapter. Both of these operations, upon completion, set the bit in the attention-summary register that corresponds to their respective drive units, and initiate an interrupt. We will assume that disk A's IO$_SEEK is completed first. The operation sets disk A's bit in the attention-summary register and generates the MASSBUS adapter's interrupt.

The MASSBUS adapter's interrupt dispatcher finds that the adapter is owned, and that the owner is expecting an interrupt. Therefore, the interrupt dispatcher incorrectly assumes that it is handling a data transfer interrupt, and, moreover, that this interrupt is the one for which the owner of the MBA is waiting.

As a result, the MASSBUS adapter's interrupt dispatcher returns control, through the fork block in the MASSBUS adapter owner's UCB, to the driver for disk B, even though disk B's operation has not completed. The disk B driver will now incorrectly assume that the device has completed its operation, which can cause serious problems.

15.4.2 Dispatching a Device Interrupt

Once the MASSBUS adapter's interrupt dispatcher determines the type of interrupt, it dispatches the interrupt to the driver. The interrupt dispatcher handles attention interrupts and data transfer interrupts in the same way, with one exception: on an attention interrupt, the interrupt dispatcher clears the MASSBUS adapter's status register (MBA$L_SR) before dispatching the interrupt to the driver. The status register contains information used only in data transfer interrupt dispatching.

How the interrupt dispatcher dispatches the interrupt to the driver differs depending on the type of controller.

The MASSBUS adapter's interrupt dispatcher handles a solicited interrupt on a dedicated controller by transferring control to the driver through the fork block in the UCB. On unsolicited interrupts on dedicated controllers, the interrupt dispatcher calls the driver's unsolicited interrupt service routine.

On dedicated controllers, the MASSBUS adapter's interrupt dispatcher always clears the attention bit in the attention-summary register before it calls back the driver after an interrupt.

Dispatching interrupts to the driver of a device that shares its controller with several other devices differs in two ways from dispatching interrupts to the driver of a device with a dedicated controller.

First, the interrupt dispatcher never clears the attention bit. This task is left to the driver because some controllers that control more than one device use this bit to synchronize their activities, and guarantee the integrity of device registers only while the bit is set. If the interrupt dispatcher clears the bit before returning control to the driver, the driver can no longer rely on the contents of the device's registers.

Second, a controller that controls several devices needs another interrupt dispatcher to handle simultaneous requests from its several subunits. This second-level interrupt dispatcher resides in the driver. After an interrupt, the MASSBUS adapter's interrupt dispatcher indirectly calls this second driver's interrupt dispatcher using code in the controller's CRB. The driver-loading procedure installs this code when it establishes the I/O database.

15.5 Special Considerations for MASSBUS Device Drivers

MASSBUS adapter considerations affect a driver's device unit initialization routine, start-I/O routines and, for multiunit controllers only, the driver's use of the DPTAB macro. MBA considerations also affect interrupt handling, as described in Section 15.4.2. The next sections in this chapter discuss programming details for writing a MASSBUS device driver.

15.5.1 Unit Initialization Routine

All drivers for MASSBUS adapter devices initialize two fields in the UCB (as well as initializing device-specific fields): UCB$B_SLAVE and UCB$B_SLAVE+1. The first of these fields should contain the controller's MASSBUS adapter unit number, which marks the controller's position on the MASSBUS adapter. The second of these contains the offset, in longwords, from the start of the MASSBUS adapter's external registers to this controller's device registers. The value of this longword offset is always 32 times the unit number of the controller.

Initialization of a device attached to a dedicated controller is simple because the device unit number and the controller position number on the MASSBUS adapter are always equal. To initialize the field UCB$B_SLAVE, copy to it the contents of UCB$W_UNIT. To initialize UCB$B_SLAVE+1, multiply the contents of UCB$W_UNIT by 32. The driver's fork process or interrupt service routine later uses this information to compute a pointer to this device's registers. By convention, R4 points to the MASSBUS adapter configuration register, and R5 points to the UCB of this device.

Thus, the following two instructions cause R3 to point to the device registers during normal system operation:

```
MOVZBL    UCB$B_SLAVE+1(R5),R3
MOVAL     MBA$L_ERB(R4)[R3],R3
```

For devices connected to a controller that controls several devices, determination of the controller's MBA position is more complex. When the unit initialization routine is invoked, the following values are in the following registers:

R3 Address of controller's device registers

R4 Address of the MBA's configuration register

R5 Address of device's UCB

The driver computes the MBA position of the controller by using R3 and R4 to determine the number of bytes from the start of the MBA's external registers to the start of the device's device registers. The difference, when divided by 128, is the controller's MBA position number.

15.5.2 The MASSBUS Adapter and the I/O Database

The UCB of a device connected to a single-unit controller, at offset UCB$L_CRB, contains the address of the MASSBUS adapter's CRB. This CRB in turn contains, at offset CRB$L_INTD+VEC$L_IDB, the address of the MASSBUS IDB. This IDB points to the base address of the MASSBUS adapter registers at offset IDB$L_CSR.

A controller that controls several devices maintains a more complicated I/O database. The device UCB, at offset UCB$L_CRB, points to the controller's CRB, and this structure points to the CRB for the MASSBUS adapter at offset CRB$L_LINK. Also, the controller's CRB points to its own IDB at offset CRB$L_INTD+VEC$L_IDB. This IDB points to the controller's device registers at offset IDB$L_CSR.

Thus, the UCB for a device always points to that device's primary CRB, whether it is the MASSBUS adapter's CRB or the controller's CRB. The primary CRB points to the secondary CRB, if one exists for the device.

Figure 15–6 shows these relationships among I/O data structures.

15.5.3 Start-I/O Routine

Depending on the function being executed, the start-I/O routine for a MASSBUS device performs all or some of the following tasks:

- Requests, if necessary, controller data channel(s) as described in Section 15.5.3.1
- Clears errors on the MASSBUS adapter by placing the value –1 into the MBA's status register; this is a write-ones-to-clear register (MASSBUS device registers and the MBA's registers are all longwords)
- Invokes the LOADMBA macro to load the MBA's map registers as described in Section 15.5.3.2
- Loads device registers to start the function
- Waits for a device interrupt or timeout
- Releases, if necessary, controller data channel(s) as described in Section 15.5.3.3
- Finishes the request like other drivers

15.5.3.1 Requesting Controller Data Channels

Device drivers for MASSBUS devices must request and receive ownership of the MASSBUS adapter channel before loading the MBA's internal registers or map registers. In addition, drivers for devices connected to multiunit controllers must obtain ownership of the controller channel before modifying the contents of controller registers that can be shared among the units connected to the controller.

Drivers for dedicated controllers must request ownership of the MASSBUS adapter channel by invoking the macro REQPCHAN.

Device drivers for controllers that control several devices invoke the REQPCHAN macro when the operation requires ownership of only the primary channel (the controller's channel). However, if the operation requires ownership of both primary and secondary channels (a data transfer operation), the driver must first obtain the controller channel and then request the MASSBUS adapter channel by invoking the REQSCHAN macro.

Again, the driver needs ownership of both channels only when performing a data transfer, and must release the channels before initiating a nondata transfer. Thus, a driver must obtain ownership of the MASSBUS adapter channel some time before initiating a data transfer and must either not own the channel or release such ownership before it invokes the WFIKPCH macro, or issue the WFIRLCH macro, following the start of a nondata transfer operation.

15.5.3.2 Loading Map Registers

MASSBUS device drivers invoke the LOADMBA macro before they initiate a data transfer, to load the MBA's map registers, the MBA's virtual-address register (MBA$L_VAR), and the MBA's byte-count register (MBA$L_BCR). Drivers cannot modify these registers during a transfer. The LOADMBA macro expects the following register contents:

- The address of the MBA's configuration register (MBA$L_CSR) in R4
- The address of the device UCB in R5

LOADMBA preserves the contents of R3 but modifies R0 through R2. The macro performs the following steps:

1. Uses the contents of UCB$W_BCNT and UCB$W_BOFF to determine the number of pages that contain pieces of the I/O buffer.
2. Beginning with the page-table entry to which UCB$L_SVAPTE points and continuing for the number of page-table entries determined in step 1, copies the page-frame numbers from the page-table entries to the corresponding map registers, starting at map register 0.
3. Ensures that the valid bit is clear in the map register that immediately follows the last map register loaded with a PFN. This prevents a hardware fault or prefetch from modifying memory.
4. Moves the negative value of the transfer byte count (UCB$W_BCNT) into the MBA's byte-count register (MBA$L_BCR).

5 Moves the byte offset in the first page of the transfer (UCB$W_BOFF) into the MBA's virtual-address register (MBA$L_VAR).

6 Returns to the start-I/O routine that invoked it.

If the I/O operation about to be initiated by the driver is a reverse operation (a read-reverse on tape), the driver must modify the contents of the MBA's virtual-address register set up by LOADMBA. Because reverse operations access the I/O buffer from its highest address through its lowest address, the value to be loaded into the MBA's virtual-address register must be the virtual address, in MBA's virtual memory, of the last byte of the buffer. This number is equal to one less than the sum of the contents of UCB$W_BOFF and UCB$W_BCNT.

15.5.3.3 Releasing Controller Data Channels

The driver releases the controller data channels by invoking the RELCHAN macro. RELCHAN releases all controller channels (both primary and secondary) currently owned by the device. To release only the secondary channel and retain ownership of the primary channel, a driver can invoke the RELSCHAN macro.

15.5.4 DPTAB Macro

The device driver for a MASSBUS device that shares its controller with other devices must set the DPT$V_SUBCNTRL bit in the **flags** argument of the DPTAB macro. Setting this bit causes the driver-loading procedure to create a second CRB and an IDB for the controller.

15.6 Interrupt Service Routines for MASSBUS Devices

The MASSBUS interrupt dispatcher (MBA$INT) gains control when it receives an interrupt from the MASSBUS adapter. Because data transfers in progress suppress attention interrupts on the MASSBUS adapter, and because several devices can request attention simultaneously, several device drivers might need to be informed of the interrupt.

MBA$INT determines which drivers should be invoked as a result of the interrupt and then passes control to these drivers. For data transfer interrupts, MBA$INT preserves the value contained in the MBA's status register at the time of the interrupt so that the driver can have access to this value.

For I/O operations that involve no data transfer, MBA$INT clears this register before invoking the driver. MBA$INT only preserves the contents of registers R2 through R5. Drivers that use other registers must save the contents of those registers, and must restore them before exiting from the interrupt service routine.

15.6.1 Transferring Control to the Interrupt Service Routine

The method by which MBA$INT invokes a driver depends upon whether the driver serves a device connected to a dedicated controller or a device that shares its controller with several other devices. Furthermore, if the device is connected to a dedicated controller, the method of transfer from MBA$INT to the driver depends upon whether or not the interrupt is expected.

For a device on a dedicated controller whose driver is expecting an interrupt, MBA$INT restores the driver context saved in the UCB fork block and transfers control (using a JSB instruction) to the instruction that follows the wait-for-interrupt instruction.

For a device on a dedicated controller whose driver is not expecting interrupts, MBA$INT obtains the address of the driver's unsolicited interrupt service routine from the driver dispatch table and calls the routine.

For a device that shares its controller with several other devices, MBA$INT transfers control to the driver's interrupt service routine by simulating a direct transfer, through an interrupt vector, to the controller's CRB. The CRB contains code that transfers control to the interrupt service routine.

MBA$INT first pushes the processor status longword (PSL) onto the stack. The routine then calls (with a JSB instruction that leaves an address within MBA$INT on the stack) the code within the CRB. This code contains the following sequence of instructions, where XX$INT is the address of the interrupt service routine and XX$IDB is the address of the controller's IDB:

```
PUSHR     #^M<R2,R3,R4,R5>
JSB       XX$INT
.LONG     XX$IDB
```

The execution of the previous instruction sequence, plus the instructions executed by MBA$INT (the pushing of the PSL onto the stack and the JSB), places a simulated interrupt frame onto the stack, including a saved PSL, a saved PC, saved registers, and a pointer to an address in the IDB.

15.6.2 Returning Control to MBA$INT

The way in which a driver returns control to MBA$INT depends on the way in which MBA$INT invoked it. Drivers for dedicated controller devices return to MBA$INT through an RSB instruction, although the RSB can execute as a result of the driver's invoking the IOFORK macro.

Drivers of devices that share a controller return control to MBA$INT by removing the indirect pointer to the IDB from the top of the stack, restoring registers R2 through R5, and executing an REI instruction. This sequence, executed within the driver's interrupt service routine, eliminates the simulated interrupt frame from the stack before returning to MBA$INT.

15.6.3 Considerations for Interrupt Service Routines

Drivers for dedicated controller devices attached to the MASSBUS do not have interrupt service routines. Instead, MBA$INT handles all the functions that a driver interrupt service routine normally provides.

Drivers of devices that share a controller on the MASSBUS must have their own interrupt service routines. In general, these routines perform the same functions as the interrupt service routines for UNIBUS and Q22 bus devices (discussed in Chapter 9). However, the two types of drivers diverge in two areas.

One difference between UNIBUS/Q22 bus and MASSBUS drivers concerns the number of registers saved by the interrupt service routine. When the interrupt dispatcher transfers control to a MASSBUS driver interrupt service routine, registers R2 through R5 are pushed onto the stack. UNIBUS/Q22 bus drivers save R0 through R5.

After handling an interrupt, both MASSBUS and UNIBUS/Q22 bus driver interrupt service routines execute an REI instruction. For UNIBUS/Q22 bus devices, the REI dismisses a real interrupt, whereas the MASSBUS driver's REI returns control to MBA$INT.

16 Generic VAXBI Device Support

This chapter provides information needed to write and load a device driver for a non-Digital-supplied device attached to the VAXBI bus. VMS provides special support for such devices in the system initialization routines for the VAX 82*x*0/83*x*0, VAX 85*x*0/8700/88*x*0, VAX 6000-series, and VAX 9000-series systems. Because of the many and varied implementations of VAXBI devices, however, VMS support must of necessity be very general. Some devices may more fully utilize the VAXBI interface than others; a device may incorporate its interface initialization logic in microcode, whereas another may defer initialization to code in its driver.

The *VAXBI Options Handbook* includes a description and guidelines for possible VAXBI device implementations. Refer to that manual for further discussion of all VAXBI topics discussed in brief in Section 16.2 and elsewhere in this chapter.

16.1 Overview

A VAXBI device driver refers to the same data structures and contains the same routines as a traditional VMS driver. A VAXBI device driver deviates from the traditional VMS driver almost exclusively in code that initializes the VAXBI interface or supports direct-memory-access (DMA) transfers for devices that address memory across the VAXBI bus. Section 16.5 discusses tasks that drivers of various VAXBI devices may perform in their initialization routines to supplement VMS initialization and that initialization performed by device microcode. Section 16.6 contains a general discussion of how some VAXBI devices and their drivers manage DMA transactions.

Section 16.4 describes those data structures the VMS adapter initialization routine creates and prepares for a generic VAXBI device, while Section 16.9 discusses the method by which its driver can be loaded into the operating system. The final section of this chapter provides reference material and includes a description of the backplane interconnect interface chip (BIIC) registers.

16.2 VAXBI Concepts

The VAXBI serves as an I/O bus for the VAX 82*x*0/83*x*0, VAX 85*x*0/8700 /88*x*0, VAX 6000-series, and VAX 9000-series systems (see Figure 1–3). The VAXBI is also the system bus for the VAX 82*x*0/83*x*0 systems. The VAX 82*x*0/83*x*0 systems have a single VAXBI; the VAX 85*x*0/8700/88*x*0, VAX 6000-series, and VAX 9000-series systems can have multiple VAXBI buses.

Each location on a VAXBI bus is called a **node**. A single VAXBI bus can service 16 nodes. In the case of the VAX 82*x*0/83*x*0 systems, these nodes can be processors, memory, and adapters; the VAX 85*x*0/8700/88*x*0 and VAX 6000-series systems permit only adapters to be attached to the VAXBI bus.[1] A node receives its **node ID**, a number from 0 to 15, from a plug on the VAXBI backplane slot into which the node module is inserted.

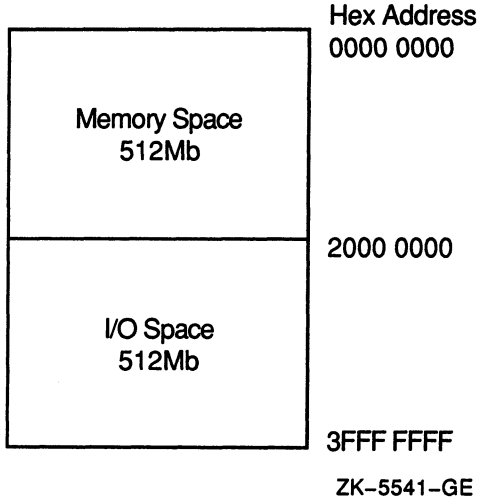
An **adapter** is a node that connects other buses, communication lines, and peripheral devices to the VAXBI bus. This chapter uses the term **device** to refer to a device or combination of devices serviced by a single adapter or controller.

16.2.1 VAXBI Address Space

Each VAXBI bus supports 30-bit addressing capability. This gigabyte of physical address space is split equally between memory and I/O address space, as shown in Figure 16–1.

All memory locations on a VAXBI bus are addressed using physical addresses in VAXBI memory space (from 00000000_{16} through $1FFFFFFF_{16}$). A VAXBI device that accesses memory directly (or indirectly through a memory-interconnect-to-VAXBI adapter), or its driver, must perform virtual-to-physical translation before transmitting a memory address on the bus. (See Section 16.6 for additional information.)

Figure 16–1 VAXBI Address Space



[1] For VAX 85*x*0/8700/88*x*0, VAX 6000-series, and VAX 9000-series systems, the memory-interconnect-to-VAXBI adapter (NBI, PBI, or DWMBA) or, more specifically, the NBIB, PBIB, or DWMBA/B resides at a node on a VAXBI bus, monitoring and controlling transactions to the memory interconnect (NMI, NMIs, or XMI) where the processors and memory reside.

VAXBI I/O address space (physical addresses 20000000_{16} through $3FFFFFFF_{16}$) is partitioned as illustrated in Figure 16–2. Figure 16–3 shows the structure of a VAXBI I/O-space address.

Figure 16–2 Description of VAXBI I/O Address Space

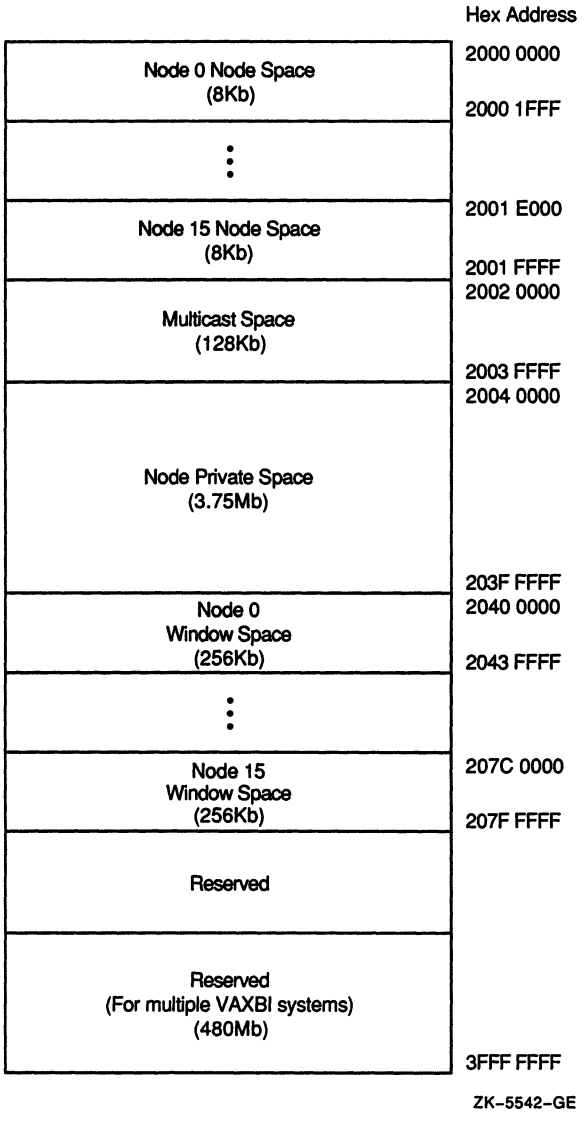
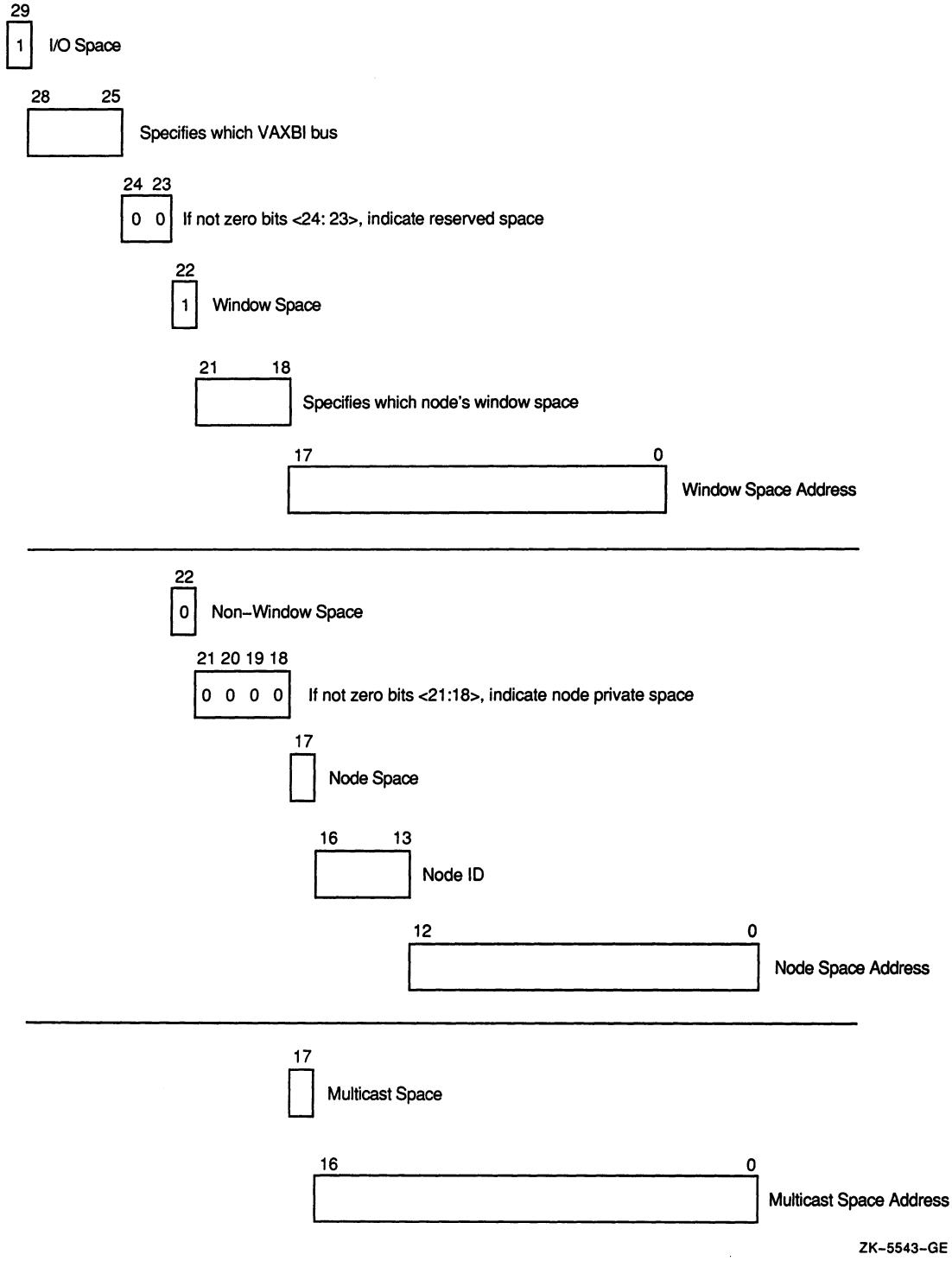


Figure 16–3 Physical Addresses in VAXBI I/O Address Space



As shown in Figures 16–2 and 16–3, VAXBI architecture grants each of the 16 nodes on a VAXBI bus two discrete sections in I/O address space.

Node space — An 8KB block of addresses consisting of 256 bytes of **BIIC CSR space**, followed by **user interface CSR space**. A device can access the control and status registers (CSRs) of its backplane interconnect interface chip by using BIIC CSR space addresses. Device-specific registers reside in user interface CSR space.

Because the VMS adapter initialization routine virtually maps node space for each VAXBI node on each VAXBI bus, a device driver can access both BIIC registers and device registers using virtual addresses. (See Sections 16.5 and 16.6 for a discussion of driver access to registers.)

Window space — A 256KB block used by a VAXBI adapter to map an I/O transfer to a target bus. Because VMS does not automatically map window space to virtual addresses, a driver that manipulates addresses in window space must itself allocate and fill sufficient system page-table entries for the range of its window space addresses. (See Section 16.5.)

Note that *node private space* contains locations used for the storage of bootstrap firmware and software. VAXBI nodes are not permitted to issue or respond to VAXBI transactions targeting locations in node private space.

16.2.2 Backplane Interconnect Interface Chip (BIIC)

The **backplane interconnect interface chip** (BIIC) serves as the primary interface between the VAXBI bus and the user interface logic of a node. The BIIC supplies the logic necessary for a node to initiate and respond to transactions on the VAXBI bus, arbitrate bus ownership, send and receive interrupt requests, and monitor bus errors.

A node can enable, control, and monitor such activities by accessing the set of BIIC registers located in the first 256 bytes of its node space. Because the VMS adapter initialization routine virtually maps node space addresses, drivers for VAXBI devices can use virtual addresses to access BIIC registers. In addition, given the virtual address of the base of a device's node space, a driver can use the symbolic offsets, masks, and bit fields defined by the VMS macro $BIICDEF (in SYS$LIBRARY:LIB.MLB). Table 16–1 in Section 16.10 describes these symbols.

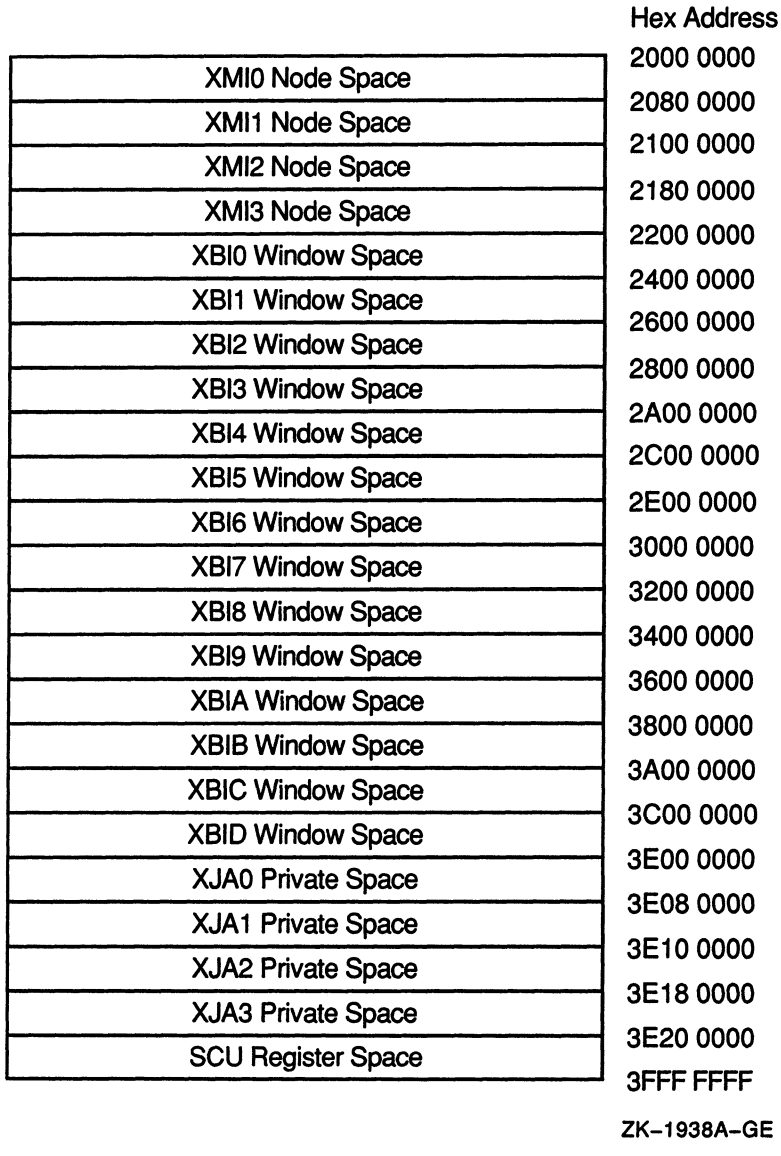
16.3 SCU/XMI Concepts

As shown in Figure 1–4, the XMI bus of an SCU/XMI system, such as a VAX 9000, serves as the primary I/O bus. Each XMI bus supports 30-bit addressing and provides 1 gigabyte of physical address space. Like the VAXBI address space, the total VAX 9000 address space is divided into equal areas of memory and I/O address space, as shown in Figure 16–1.

All memory locations on a VAX 9000 XMI bus are addressed using physical addresses in XMI memory space (from $0000\ 0000_{16}$ through $1FFF\ FFFF_{16}$).

VAX 9000 XMI I/O address space (physical addresses 2000 0000_{16} through 3FFF $FFFF_{16}$) is partitioned as illustrated in Figure 16–4.

Figure 16–4 SCU/XMI Systems I/O Address Space



The assignment of I/O addresses, shown in Figure 16–4, for any VAX 9000 system supports two levels of bus structure: the XMI and the VAXBI. A VAX 9000 system uses the XMI as an I/O bus and may have up to 4 XMIs, depending on the model. Each XMI can have up to 12 nodes or devices numbered 1 through D hexadecimal. (The XJA adapter occupies node 0 on an XMI.)

The four XMI-bus node spaces (XMI0 to XMI3) are assigned the first region of XMI I/O address space. By means of address translation performed by the system control unit (SCU) and the I/O control unit, CPUs address individual XMI device CSRs using XMI nodespace.

Fourteen XBI window space regions follow XMI nodespace, one for each DWMBA adapter that might be present on any of the XMI buses. A DWMBA is an I/O adapter that connects a VAXBI bus to an XMI bus. Each DWMBA (14 maximum) is physically mapped into its own XBI window space (XBIx) in the I/O block. The VAXBI window spaces are named XBI0 to $XBID_{16}$, and span the XBI window space region. XBI window space allows CPUs to address the individual VAXBI device CSRs.

In XMI I/O space, a given XBI's window space is determined by the XBI's XMI node number. There is a limit of 8 XBIs on a given XMI bus and a limit of 14 XBIs across all XMI buses. In the assignment of XBI window spaces, the XBI with the lowest node number on XMI0 is assigned to XBI0 window space. The XBI with the second lowest node number in XMI0 is assigned to XBI1 window space, and so on through all XBIs on XMI0; then to XMI1, XMI2, and until XMI3 is exhausted or the 14th XBI is found.

An XJA of the SCU/XMI bus architecture is an adapter that connects the SCU ports to the XMI bus. The XJAs have a private space region in the I/O block that allows CPUs to address the XJA CSRs. Since there are up to four XMIs, there can be four XJAs. In the XJA private space region, XJAs are mapped XJA0 through XJA3.

16.4 Initialization Performed by VMS

During the phase of system initialization known as adapter initialization VMS performs a set of system-specific tasks to identify and configure each device it discovers at each of the 16 nodes on each VAXBI bus in the system configuration.

The adapter initialization module configures Digital-supplied and non-Digital-supplied devices alike, performing the following activities as part of its initialization cycle:

1. Tests for the presence of a device at the node by issuing a MOVL instruction, the target of which is a system virtual address temporarily mapped to the first longword of its node space. If this instruction is successful, it returns the contents of the BIIC Device Type Register of the addressed node to the processor.[2]
2. Records the contents of the low 16 bits of the BIIC Device Type Register, plus an I/O bus identifier in the slot in the CONFREGL array that corresponds to the VAXBI bus and node at which it found

[2] If no device exists at a given VAXBI node address, the CPU becomes aware of this in a system-specific way. For example, the VAX 82*x*0/83*x*0 systems experience a machine check, whereas the VAX 85*x*0/8700/88*x*0, VAX 6000-series, and VAX 9000-series systems determine that the node is vacant by reading an NXM (nonexistent memory) error from the BIIC Bus Error Register of the NBIB, PBIB, or DWMBA on the VAXBI being examined.

the device,[3] and compares this value against a table of recognized device types.

3 If it *recognizes* the device, maps the number of pages specified in the table for the device type, and places the system virtual address of the base of the mapped node space in the slot in the SBICONF array that corresponds to the VAXBI bus and node at which it found the device.[4]

If it does *not* recognize the device, maps the entire 8KB of the node's node space into VMS virtual address space by allocating 16 system page-table entries (SPTEs) and associating them with the 16 page-frame numbers (PFNs) of the physical addresses assigned to this node's node space on this VAXBI bus. The adapter initialization module then saves the base system virtual address of the resulting 8KB range in the longword slot corresponding to this node in the SBICONF array.

4 Performs such additional tasks as allocating and filling in data structures in a device-specific manner. For a non-Digital-supplied device attached to a VAXBI bus, VMS creates generic versions of the channel request block, interrupt dispatch block, and adapter control block—and fills in the appropriate vectors in the system control block—as discussed in Section 16.4.1.

For devices it *does* recognize, VMS additionally calls a VMS-supplied subroutine, the address of which it obtains from the device-type table, that performs further device-specific initialization.

For devices it does *not* recognize, VMS must defer device-specific initialization to the device driver's initialization routine.

16.4.1 Data Structures

The adapter initialization module creates and prepares a channel request block, interrupt dispatch block, and an adapter control block in the manner described in this section. For each data structure it creates, VMS fills in the first three longwords with the standard VMS header information (that is, the structure type, size, and links).

Channel Request Block

For the newly created channel request block (CRB), VMS performs the following tasks:

- Sets up the resource wait queue header (CRB$L_WQFL and CRB$L_WQBL)

[3] The CONFREGL array is a set of longwords in system pool pointed to by EXE$GL_CONFREGL. The CONFREGL array contains an entry for each possible VAXBI node. For VAX 82*x*0/83*x*0 systems, with one VAXBI, this array has 16 entries. For VAX 85*x*0/8700/88*x*0, VAX 6000-series, and VAX 9000-series systems, this array has 16 entries for each VAXBI bus on the system.

[4] The SBICONF array is a set of longwords, similar in structure to the CONFREGL array and pointed to by MMG$GL_SBICONF, that lists the system virtual addresses of the base of the node space for each node on a VAXBI bus.

- Sets the bit CRB$V_UNINIT in CRB$B_MASK to indicate to the System Generation Utility that, although the CRB exists, its controller initialization routine has not yet been called
- Initializes four interrupt dispatchers (CRBL_INTD, CRBL_INTD2, and so on) so that they have the effect of pushing general registers R0 through R5 onto the stack, and issuing a JSB instruction

The adapter initialization module always creates the four vectors, in contrast to the methods by which UNIBUS/Q22 bus drivers control the number of vectors created (see Section 14.3.3). The destination of the JSB instruction at initialization is a standard null interrupt handler which merely dismisses the interrupt. Later, when the specific device driver is loaded for the device (see Section 16.9), the driver's interrupt service routine address replaces this null interrupt handler in the dispatchers. As necessary, the driver specifies the addresses of its interrupt service routines as follows:

```
DPT_STORE,CRB,CRB$L_INTD,D,isr_for_1st_vector
DPT_STORE,CRB,CRB$L_INTD2+VEC$L_ISR,D,isr_for_2nd_vector
DPT_STORE,CRB,CRB$L_INTD+(2*VEC$K_LENGTH)+VEC$L_ISR,D,isr_for_3rd_vector
DPT_STORE,CRB,CRB$L_INTD+(3*VEC$K_LENGTH)+VEC$L_ISR,D,isr_for_4th_vector
```

Interrupt Dispatch Block

VMS initializes the interrupt dispatch block (IDB) in the following manner:

- Sets the number of device units controlled by this interrupt dispatch block (IDB$W_UNITS) to 1. The list of unit control block (UCB) addresses in this IDB, as a result, is one longword in size. The driver-loading procedure writes a UCB address into this longword whenever it creates a new UCB associated with the controller. Because there is only one slot in this array, drivers for non-Digital-supplied multiunit controllers must use a different mechanism to locate the UCB of interest at the time of an interrupt.
- Copies the virtual address of the base of this device's node space to IDB$L_CSR from the corresponding slot in the SBICONF array.

Adapter Control Block

VMS creates a truncated adapter control block (ADP) for a non-Digital-supplied VAXBI device (48 bytes as opposed to the traditional 600 bytes). The ADP it creates contains no fields reserved for the allocation and accounting of data paths or map registers. VMS prepares this generic ADP in the following manner:

- Copies the virtual address of the base of this device's node space to ADP$L_CSR from IDB$L_CSR.
- Places the VAXBI node ID of this device in ADP$W_TR.
- Stores the value AT$_GENBI (signifying the *generic* VAXBI ADP type) in ADP$W_ADPTYPE.

- Calculates the address of the first of the four interrupt vectors for this node in the system control block (SCB), and places it in ADP$L_AVECTOR. A driver can determine the addresses of the other three SCB vectors by adding 64, 128, or 192, respectively, to the address of this first SCB vector.
- Saves the offset of this first SCB vector from the start of its SCB page in ADP$W_BI_VECTOR. (Refer to Section 16.4.2 for a description of the SCB.)
- Places in ADP$L_BI_IDR a longword mask with a single bit set, as appropriate to the VAX system, that specifies which VAXBI node should become the destination of interrupts from this node. In VAX 82*x*0/83*x*0 systems, the VAXBI node of the primary processor becomes the destination for interrupts. In VAX 85*x*0/8700/88*x*0, VAX 6000-series, and VAX 9000-series systems, it is the VAXBI node of the NBIB, PBIB, or DWMBA/B on the particular VAXBI bus on which this device resides that becomes the destination for such interrupts.
- Stores in ADP$L_MBASCB—and in each of the device's four SCB vectors—the address of the interrupt dispatcher. The actual stored value is CRBL_INTD+1, CRBL_INTD2, and so on, the set low bit of the address indicating that the interrupt stack be used to service the interrupt. Certain powerfail recovery operations use the contents of ADP$L_MBASCB to refresh the SCB vectors.
- Saves in ADP$L_MBASPTE the contents of the first of the 16 SPTEs that map the device's node space. Certain recovery operations use the contents of ADP$L_MBASPTE to restore correct SPTE values and remap node space following a power failure.
- Places in ADP$L_BIMASTER the address of the ADP of the memory-interconnect-to-VAXBI-adapter (NBI, PBI, or DWMBA). Note that there is no memory-interconnect-to-VAXBI adapter for VAX 82*x*0/83*x*0 configurations.

16.4.2 System Control Block

The system control block (SCB) consists of one or more pages of vectors. For all VAX processing systems, the first half page contains vectors used in exception dispatching. VMS uses the remainder of the first page, as well as subsequent pages, in a system-specific way.

For VAX 82*x*0/83*x*0 systems, VMS assigns the vectors from 100_{16} to $1FC_{16}$ to VAXBI devices in the order of their node IDs.

The VAX 85*x*0/8700/88*x*0 system architectures relegate vectors 100_{16} to $1FC_{16}$ to NMI nexus vectors in page 0. Page 1 is reserved for the first "offsettable" device that exists in the system. (An "offsettable" device is an adapter such as the VAXBI-to-UNIBUS adapter (DWBUA or DWMUA) that passes interrupts from devices on another bus to the VAXBI and, from there, to the memory interconnect (NMI or XMI) and the processor.) If there is more than one "offsettable" device, an additional SCB page is needed for each.

Ultimately, the vectors for other devices attached to each of the six possible VAXBI buses of the system are contained in the six corresponding SCB pages from page 26 to page 31. In a 4-VAXBI system, for instance, vectors for devices connected to VAXBI 0 and VAXBI 1 on NBI/PBI/DWMBA 0 are assigned to pages 28 and 29 of the SCB, respectively; vectors for devices connected to VAXBI 0 and VAXBI 1 on NBI/PBI/DWMBA 1 are likewise assigned to pages 30 and 31. In a 6-VAXBI system, the vectors are assigned in a similar fashion, starting at page 26.

For VAX 6000-series systems, the vectors for XMI devices are assigned from 100_{16} to $1FC_{16}$ in the first SCB page (page 0). Then vectors for devices connected to VAXBI 0 start on page 1 and run from 300_{16} to $3FC_{16}$, then devices for VAXBI 1 are assigned vectors (500_{16} to $5FC_{16}$) in page 2, and so on for the remaining VAXBIs and respective pages.

For VAX 9000-series systems, the vectors for devices connected to XMI0 bus are assigned from 100_{16} to $1FC_{16}$ in SCB page 0. Then vectors for devices connected to XMI1 bus start on page 1 and run from 300_{16} to $3FC_{16}$, and so on for devices on buses up through XMI3 bus. Then, vectors for devices connected to VAXBI 0 start on page 4 with vectors $B00_{16}$ to BFC_{16}, and so on for the remaining VAXBI buses and respective pages.

Generally, a VAX processor obtains a device vector from the BIIC registers of the node that has requested the interrupt (see Figure 16–5). Information supplied in the device vector allows the processor to index to the corresponding interrupt-dispatching vector in the appropriate page of the SCB. For VAX 82*x*0/83*x*0 systems, such information includes the interrupt level of the device and its VAXBI node ID. A similar vector for VAX 85*x*0/8700/88*x*0, VAX 6000 series and VAX 9000-series devices further specifies the appropriate NBI/PBI/DWMBA vector offset and the number of the VAXBI bus.

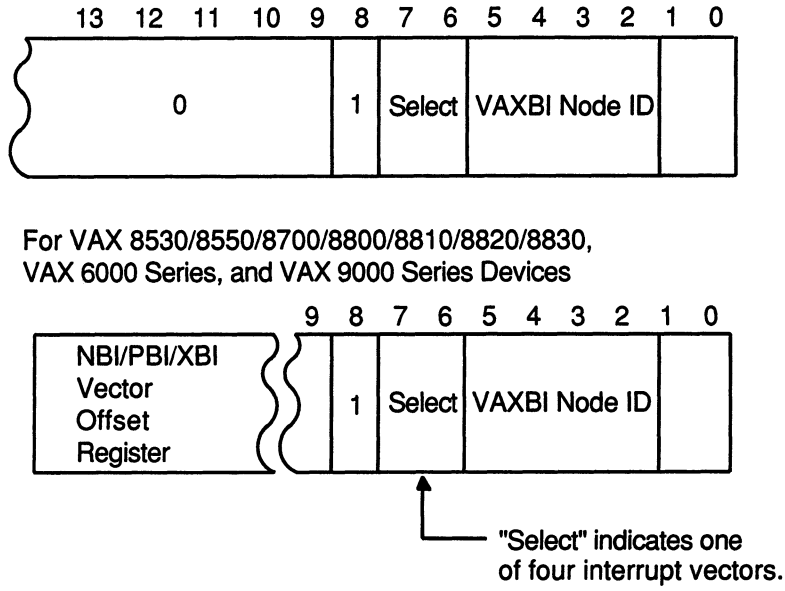
The specific SCB interrupt-dispatching vector, thus found, transfers control to the interrupt-dispatching code in the device's CRB. Upon an interrupt from this device, the SCB vector directs flow into the interrupt dispatcher in the CRB, which saves the register contents and dispatches to the interrupt service routine established by the device driver.

16.5 Initialization Performed by the VAXBI Device Driver

All generic VAXBI device drivers must specify *GENBI* as the adapter type in the **adapter** argument to the DPTAB macro.

The device driver's initialization routines are expected to initialize the device-specific aspects of the VAXBI device. For non-Digital-supplied devices, the initialization routines perform the sort of tasks that the adapter initialization module performs for the Digital-supplied devices it discovers on a VAXBI bus. For single-unit devices, a separate unit initialization routine may not be necessary.

Figure 16–5 VAXBI Device Vectors



The VMS System Generation Utility (SYSGEN) calls the controller initialization routine at IPL 31 (see Section 11.1), passing it the following values in the listed general registers:

- R4 pointing to the system virtual address of the device's node space
- R5 pointing to the IDB
- R6 pointing to the DDB
- R8 pointing to the CRB

After the controller initialization routine has completed, SYSGEN calls the driver's unit initialization routine at IPL 31, and passes it the following values in the listed general registers:

- R3 pointing to the system virtual address of the device's node space
- R5 pointing to the UCB

Hardware initialization might include such activities as writing values to BIIC and device-specific registers, examining the results of the BIIC self-test, mapping a node's window space, building data structures to control the device, and linking these structures into chains of similar data structures.

This section provides some ideas and guidelines for code that may be necessary in an initialization routine. There is no requirement that driver code perform all of the functions discussed here. The needs of various devices differ, and some devices make more demands on driver software than others.

Code examples in the section assume that R4 initially contains the virtual address of the base of the device's node space and R8 contains the virtual address of the device's CRB.

16.5.1 Examining BIIC Self-Test Status

According to the hardware specification for all devices attached to a VAXBI bus, a VAXBI node undergoes a self-test on power failure recovery and at system boot time. The BIIC indicates the successful completion of the self-test by setting BIIC$V_STS and by clearing BIIC$V_BROKE in BIIC$L_BICSR.

A driver unit initialization routine should test these bits before performing any transaction on the VAXBI bus. If BIIC$V_STS is clear, then self-test is still under way. If BIIC$V_BROKE is set, then the driver action is implementation-specific. In any event, a driver should not set UCB$V_ONLINE in UCB$L_STS if the node is not usable.

The maximum duration of the BIIC self-test is 10 seconds. If a VAXBI node implements the maximum self-test time, then the driver unit initialization routine may have to spin wait for the setting of BIIC$V_STS (for instance, by embedding the testing instructions in an invocation of the TIMEDWAIT macro). Driver unit initialization routines should perform this spin wait only when UCB$V_POWER in UCB$L_STS is set. Otherwise, the driver is being loaded by SYSGEN, and a long spin wait at high IPL will have adverse effects on the rest of the VMS system.

Normally, only diagnostics initiate a self-test by setting the SST bit in the BIIC. A VAXBI driver that sets this bit must take special precautions to avoid a machine check and to avoid undetected corruption of VAXBI memory. These precautions include the following steps:

1 Use the $PRTCTINI macro to begin a machine check protection block, supplying the location of the end of the block in the **label** argument and the mask value #<MCHK$M_NEXM!MCHK$M_LOG> in the **mask** argument. (Note that you must include an invocation of the $MCHKDEF macro in the driver to use these symbols.) Code within the block executes at IPL 31.

2 Invoke the BI_NODE_RESET macro as follows:

```
BI_NODE_RESET  CSR=R4
```

The BI_NODE_RESET macro uses the recommended instruction sequence to disable arbitration on the VAXBI node to be reset, and sets the node reset and self-test status bits in BIIC$L_BICSR. The use of any instruction sequence other than that defined by the BI_NODE_RESET macro to perform these actions may cause an undefined condition on the VAXBI bus.

3 Use the $PRTCTEND macro to end the machine check protection block. You must specify in the **label** argument the same value you specified in the **label** argument to the $PRTCTINI macro.

4 Do not access the BIIC registers for at least 1 millisecond. You may not even check the state of the STS bit during this interval.

5 Do not access any other address on the VAXBI node until the self-test has completed.

Normally, only diagnostics initiate a self-test by setting the SST bit in the BIIC. A VAXBI driver that sets this bit must take special precautions to avoid a machine check and to avoid undetected corruption of VAXBI memory. These precautions include the following steps:

1 Use the $PRTCTINI macro to begin a machine check protection block, supplying the location of the end of the block in the **label** argument and the mask value #<MCHK$M_NEXM!MCHK$M_LOG> in the **mask** argument. (Note that you must include an invocation of the $MCHKDEF macro in the driver to use these symbols.) Code within the block executes at IPL 31.

2 Disable arbitration on the VAXBI node being reset by setting BIIC$V_ARBCNTRL in BIIC$L_BICSR.

3 Set BIIC$V_SST and BIIC$V_STS simultaneously to initiate the self-test. Do not set BIIC$V_SST in the same instruction that disables arbitration.

4 Use the $PRTCTEND macro to end the machine check protection block. You must specify in the **label** argument the same value you specified in the **label** argument to the $PRTCTINI macro.

5 Do not access the BIIC registers for at least 1 millisecond. You cannot even check the state of the STS bit during this interval.

6 Do not access any other address on the VAXBI node until the self-test has completed.

16.5.2 Clearing BIIC Errors, Setting Interrupts, and Enabling Interrupts

There is a set of tasks that a VAXBI driver should perform during initialization that ensures that interrupts are properly enabled and delivered to an appropriate VAXBI target node. These tasks include the following:

- Clearing any outstanding set bits in the Bus Error Register.
- Setting the target node for interrupts in the Interrupt Destination Register.
- Setting the device interrupt vector in the Error Interrupt Control Register.
- Setting the device interrupt vector in the User Interface Interrupt Control Register.
- Enabling hard and soft error interrupts as required by the device. Typically hard errors are enabled and soft errors are disabled.

- Enabling interrupts upon certain types of transactions to user interface CSR space.

It is important that the interrupt vectors and destination be set up *before* BIIC hard error and soft error interrupts are enabled. An error occurring while error interrupts are enabled but the vector is not initialized could lead to an invalid condition.

16.5.2.1 Clearing the Bus Error Register

The following example clears all set bits in the Bus Error Register (BIIC$L_BER) to prevent spurious or pending error interrupts at initialization.

```
MOVL    BIIC$L_BER(R4),-     ;Clear all set write-1-to-clear
        BIIC$L_BER(R4)       ;bits in BIIC$L_BER
```

16.5.2.2 Loading the Interrupt Destination Register

The Interrupt Destination Register (BIIC$L_IDR) specifies which VAXBI node should become the destination of interrupts from this node. In VAX 82*x*0/83*x*0 systems, the VAXBI node of the primary CPU becomes the destination for interrupts. In VAX 85*x*0/8700/88*x*0, VAX 6000-series, and VAX 9000-series systems, the VAXBI node of the NBIB, PBIB, or DWMBA /B on the particular VAXBI on which this device resides becomes the destination for such interrupts.

The VMS system initialization procedure described in Section 16.4 creates a 32-bit mask with the appropriate bit set and stores it in ADP$L_BI_IDR. If a driver must set the Interrupt Destination Register, it can simply move this value to the BIIC register:

```
MOVL    CRB$L_INTD+VEC$L_ADP(R8),R0    ;Get ADP address
MOVL    ADP$L_BI_IDR(R0),-             ;Write to IDR
            BIIC$L_IDR(R4)
```

16.5.2.3 Setting Interrupt Vectors

A VAXBI node uses the Error Interrupt Control Register (BIIC$L_EICR) to determine the SCB vector through which to interrupt when a BIIC at this node detects a bus error. The User Interface Interrupt Control Register (BIIC$L_UICR) similarly controls the operation of interrupts initiated by the device at this node. A driver can also use the Error Interrupt Control Register to support a device that generates secondary interrupt vectors.

Because the VMS system initialization procedure described in Section 16.4 saves the offset of the node's first SCB vector from the start of its SCB page in ADP$W_BI_VECTOR, a driver can initialize both of these registers by using code similar to that in the following example:

```
MOVL    CRB$L_INTD+VEC$L_ADP(R8),R0        ;Get ADP address
MOVZWL  ADP$W_BI_VECTOR(R0),R2             ;Get device vector
MOVL    BIIC$L_UICR(R4),BIIC$L_UICR(R4)    ;Clear user vector
MOVL    R2,BIIC$L_UICR(R4)                 ;Set user vector
BISL    #1@<BIIC$V_LEVEL+BIIC$S_LEVEL-1>,R2
                                           ;OR in interrupt level
                                           ;BR7 in this case
MOVL    BIIC$L_EICR(R4),BIIC$L_EICR(R4)    ;Clear error vector
MOVL    R2,BIIC$L_EICR(R4)                 ;Set error vector
```

Note that the driver clears both vectors before it actually sets them. Clearing BIIC$L_UICR and BIIC$L_EICR causes any pending interrupt to be cleared. Also note that the interrupt level must be set in BIIC$L_EICR, in this case BR7. If the level is not set, an interrupt will never be generated.

16.5.2.4 Enabling Error Interrupts

Finally, to enable interrupts that report errors detected by the node's BIIC, the controller initialization routine can set the soft error interrupt-enable or hard error interrupt-enable bits in the VAXBI Control and Status Register. The BIIC sets bits in the Bus Error Register (BIIC$L_BER) to reflect the type of bus error reported by the interrupt.

```
BISL    #<BIIC$M_SEIE!BIIC$M_HEIE>,- ;Soft error interrupt enable
            BIIC$L_BICSR(R4)          ;Hard error interrupt enable
```

16.5.2.5 Enabling BIIC Options

Device registers are in the area of node space called user interface CSR space, and are located following the 256 bytes reserved for the BIIC-required registers. Use of user interface CSR space is implementation-dependent.

For the processor to be alerted to various transactions directed at user interface CSR space, the controller initialization routine of devices that support such transactions should set appropriate bits in the BCI Control and Status Register (BIIC$L_BCICR). See Table 16–1 for definitions of these bits.

The following example enables a node to alert the node specified as the interrupt destination (in BIIC$L_IDR) when a retry timeout, STOP command, or read or write transaction is directed at its user interface CSR space.

```
BISL    #<BIIC$M_STOPEN!-          ;Stop enable
          BIIC$M_RTOEVEN!-         ;Retry timeout enable
          BIIC$M_UCSREN>,-         ;User CSR enable
          BIIC$L_BCICR(R4)
```

16.5.3 Mapping Window Space

Each VAXBI, starting at address 20400000_{16} in its I/O address space, provides 16 address blocks of 256K bytes apiece, called **window space**. VAXBI nodes can use window space if it is necessary to map VAXBI transactions to memory space on a target bus, although only such nodes as the DWBUA or DWMUA adapter currently use this feature.

Whereas the VMS initialization routine maps each VAXBI node's node space to virtual addresses, it does not automatically map each node's window space. If a device needs to use its window space, it is up to the driver's unit initialization routine to map this space.

First of all, the driver must determine the starting physical address of the node's window space. Figure 16–3 illustrates how VAXBI addresses are constructed. Drivers can use the following VMS-supplied macros (in SYS$LIBRARY:LIB.MLB) to access pertinent VAXBI addresses and values:

$IO8SSDEF (for VAX 82x0/83x0 systems)
$IO8NNDEF (for VAX 8530/8550/8700/8800 systems)
$IO8NNDEF and $IO8PSDEF (for VAX 8810/8820/8830 systems)
$IO9CCDEF (for VAX 6000-2xx/6000-3xx series systems)
$IO9RRDEF (for VAX 6000-4xx series system)
$IO9AQDEF (for VAX 9000-series system)

A driver calculates the starting address of a node's window space by first determining the offset to the start of the VAXBI node's window space from the beginning of its VAXBI I/O space. To do so, it performs the following tasks:

1 Extracts the VAXBI node ID from bits <3:0> of ADP$W_TR.

2 Multiplies the VAXBI node ID with the size of window space. The driver obtains this value from the following symbols:

Symbol	System
IO8SS$AL_NDSPER	VAX 82x0/83x0
IO8NN$AL_NDSPER	VAX 85x0/8700/88x0
IO9CC$C_BIWSIZ	VAX 6000-2xx/6000-3xx series
IO9RR$C_BIWSIZ	VAX 6000-4xx series
IO9AQ$C_BIWSIZ	VAX 9000-series

3 Adds the address of the window space of VAXBI node 0 to this value. The driver obtains this value from the following symbols:

Symbol	System
IO8SS$AL_NODESP	VAX 82x0/83x0
IO8NN$AL_NODESP	VAX 85x0/8700/88x0
IO9CC$C_BIWINDOW	VAX 6000-2xx/6000-3xx series
IO9RR$C_BIWINDOW	VAX 6000-4xx series
IO9AQ$C_BIWINDOW	VAX 9000 series

The driver for a device on a VAX system configured with more than one VAXBI bus (for instance, the VAX 85x0/8700/88x0, VAX 6000 series, or VAX 9000 series systems) must proceed to calculate the start of the I/O address space of the VAXBI bus to which the device is attached. It adds the result of the following steps to the value it has obtained in the prior steps:

1 Determine the offset to the I/O address space of the VAXBI bus to which the node is attached.

For VAX 6000-series systems, a driver first must obtain the XMI node ID of the DWMBA to which the VAXBI is connected. It determines the address of the DWMBA's ADP at offset ADP$L_BIMASTER of the node's ADP. It then finds the XMI node of the DWMBA at offset ADP$W_XBIA_TR of the ADP.

For VAX 85*x*0/8700/88*x*0 configurations, a driver obtains the VAXBI bus number from bits <7:4> from offset ADP$W_TR of the node's ADP.

For VAX 9000 systems, a driver obtains the relative VAXBI number from ADP$B_REL_BI.

2 Multiply this value by 2000000_{16}, the amount of physical address space allocated for each VAXBI bus.

3 Add to this value the base of I/O address space. The driver obtains this value from the following symbols:

Symbol	System
IO8SS$AL_IOBASE	VAX 82*x*0/83*x*0
IO8NN$AL_IOBASE	VAX 85*x*0/8700/88*x*0
IO9CC$AL_IOBASE	VAX 6000-2xx/6000-3xx series
IO9RR$AL_IOBASE	VAX 6000-4xx series
IO9AQ$AL_XBI0_WINDSP_30	VAX 9000 series

After performing these calculations, the driver must associate each page of window space to be used with a system page-table entry (SPTE) that maps the page-frame number (PFN) of the physical page in window space to a system virtual address. VMS includes the routine LDR$ALLOC_PT, described in the *VMS Device Support Reference Manual*, that allocates system page-table entries (SPTEs) for a specified number of pages.

Because LDR$ALLOC_PT executes at IPL$_SYNCH (holding the MMG spin lock in a VMS multiprocessing system), the controller initialization routine must fork from IPL$_POWER to fork IPL (using the CRB fork block) prior to calling it. See Section 11.1.5 for a discussion of forking in a driver initialization routine.

Finally, once the SPTEs have been allocated, the driver moves the PFNs of the window space pages into the SPTEs, sets their valid bits, and initializes them in a device-specific manner.

16.6 DMA Transfers

The method by which a device accomplishes direct-memory-access (DMA) transfers depends upon the characteristics of the device. As part of a VAXBI read or write transaction, such a device must place on the VAXBI bus a physical address, the target of which is a memory node or a node (such as an NBIB adapter) that transmits the request to memory across another bus.

For the DMA device to successfully access the memory pages of a buffer involved in an I/O transfer, it must be given sufficient information as to the size and location of these buffer pages, the type of transaction that is requested, an offset into the first page of the buffer, and the length of the transaction. In addition, if the size of the transaction causes it to exceed the boundaries of a page, the device must have some means of accessing the remaining pages—even if they are, as is most likely, scattered throughout physical memory.

As a result, devices make use of several types of structures, the purpose of which is to help generate a succession of contiguous physical addresses on the VAXBI bus, that map to the various pages of the buffer involved in the transfer. Some possible strategies of this sort include the following:

- A physically contiguous buffer in memory
- System page tables in system memory
- Process page tables locked in system memory
- Map registers in the device's VAXBI I/O address space

A separate but related issue results from the fact that the original buffer, as specified in the user $QIO request, is in process space and is mapped by process page-table entries. Because the driver cannot rely on process context existing at the time the device is ready to service the I/O request, it must have some means of guaranteeing that it can access both the data involved in the transfer and the page-table entries that map the buffer.

VMS supplies two separate techniques, applied by traditional VMS drivers and described in Section 6.3.1.

- **Direct I/O**, the technique used most commonly by DMA drivers, locks the user buffer in memory as well as the page-table entries that map it. Such a driver calls a VMS-supplied FDT routine that prepares the user buffer for direct I/O.
- **Buffered I/O** is the strategy whereby the driver FDT routine allocates a buffer from nonpaged pool. It is this intermediate buffer that is involved in the DMA transfer. The FDT routine copies the data from the user buffer to the system buffer for a write request; I/O postprocessing routines deliver data from the system buffer to the user buffer for a read request.

That DMA drivers may make use of either VMS direct I/O or buffered I/O is one way by which these drivers can supply specific information needed by the device to accomplish a DMA transfer. Those driver FDT routines that call a VMS direct-I/O FDT routine provide the following information in the device's unit control block (UCB):

UCB$L_SVAPTE	Virtual address of the system page-table entry (SPTE) for the first page used in the transfer
UCB$W_BOFF	Byte offset in the first page of the transfer buffer
UCB$W_BCNT	Size in bytes of the transfer

FDT routines for buffered I/O call EXE$ALLOCBUF, EXE$DEBIT_BYTCNT_ALO, or EXE$DEBIT_BYTCNT_BYTLM_ALO to obtain a nonpaged pool buffer (debiting a job's byte count quota in the last two routines) and initialize the same UCB fields with the following information:

UCB$L_SVAPTE	Virtual address of system buffer used in the I/O transfer
UCB$W_BOFF	Number of bytes to be charged to process for transfer
UCB$W_BCNT	Size in bytes of the transfer

If a driver's fork process must manipulate the data in any way at fork level (that is, outside of the driver's FDT routines), then it needs a virtual address it can use to access the data. Such a requirement could cause the driver writer to consider structuring the driver so that it uses buffered I/O. For short transfers, this need also could be accommodated by the driver's loading an SPTE with the correct PFN and computing the associated system virtual address. The drivers for the disks that have ECC correction applied by the host do this when there is an ECC error detected. The controller can tell the driver that the error in the data in memory can be corrected by applying some pattern to a part of the data, but the fork process has to perform the correction, not the controller.

```
MOVL    IRP$L_SVAPTE(R3),R2          ;Get address of system buffer
SUBW3   #12,8(R2),UCB$W_BCNT(R5)     ;Calculate system buffer
                                     ; length
BICW3   #C^<VA$M_BYTE>,(R2),UCB$W_BOFF ;Put offset in buffer
EXTZV   #VA$V_VPN,#VA$S_VPN(R2),R2   ;Get system virtual page
                                     ; number
MOVL    G^MMG$GL_SPTBASE,R1          ;Get address of system page
                                     ; table
MOVAL   (R1)[R2],UCB$L_SVAPTE(R5)    ;Get system virtual address
                                     ; of page
```

16.6.1 Example: DMB32 Asynchronous/Synchronous Multiplexer

The DMB32 asynchronous/synchronous multiplexer can use any of four different modes of address translation for DMA accesses. Under each of these modes, the DMB32 requires that its driver supply an address by which it can either directly or indirectly obtain the pages of the buffer that is involved in the transfer. The four different translation modes require such addresses in one of the following forms:

1. System virtual address of a buffer
2. System virtual address of a page-table entry
3. Physical address of a page table
4. Address of a physically-contiguous buffer

System Virtual Address of a Buffer and a Page-Table Entry

The DMB32 can itself perform the first two types of address translation because it can read entries in the VMS system page table (see the *VAX/VMS Internals and Data Structures* manual for a description of page-table entries). The controller initialization routine of a DMB32 device driver supplies the physical address and length of the VMS system page table, plus the virtual address and length of the VMS global page table. It also sets a page-table-valid bit in a device maintenance register.

As a result, a driver for a DMB32 device could use either direct I/O or buffered I/O, and accordingly load a device register with the system virtual address of the page-table entry that maps the buffer or the system virtual address of the buffer itself. After the driver has loaded other device registers with a buffer offset value and a transfer size—and set the "start" bit in a DMB32 line-control register—the DMB32 performs the transfer without any additional mapping or other driver intervention.

Physical Address of a Page Table

In this mode, the DMB32 can be given the physical address of a page table that maps the I/O transfer. The DMB32 architecture mandates that each page-table entry be four bytes long and that the page table be aligned on a longword boundary. Also, each page is 512 bytes long. However, the page table can be anywhere in memory, possibly at a range of VAXBI I/O-space addresses belonging to the node to which the DMB32 adapter is attached. To perform a DMA transfer under this addressing mode, the DMB32 adapter requires the offset of the first byte of the buffer that is in the page described by the page-table entry. Each page-table entry contains bits <29:9> of the physical address of the page that is to be accessed.

In this case, the driver must extract the PFNs of the pages involved in the transfer and insert them into the page table of the device. The following is an example of a routine that translates a system virtual address to a physical address. It returns the physical address at the top of the stack.

```
VIRT_TO_PHYAD:
    PUSHL   (SP)                    ;Create slot at top of stack
                                    ;for return value
    PUSHR   #^M<R0,R1,R2,R3>        ;Save registers
    BICL3   #-512,R1,R0             ;R0 = byte offset of address
    EXTZV   #VA$V_VPN,-             ;Extract VPN
             #VA$S_VPN,R1,R2        ; and put it in R2
    MOVL     G^MMG$GL_SPTBASE,R3    ;R3 => system page table
    MOVL    (R3)[R2],R3             ;R3 => PTE
    EXTZV   #PTE$V_PFN,-            ;Get page frame number of
             #PTE$S_PFN,R3,R3       ; buffer page into R3
    ASHL    #VA$V_VPN,R3,R3         ;Shift into place for
                                    ; physical address
    BISL3   R0,R3,20(SP)            ;Put result into stack slot
    POPR    #^M<R0,R1,R2,R3>        ;Restore registers
    RSB                             ;Return to caller
```

Physical Address of a Buffer

If the device can neither read system page tables nor has its own scatter-gather map—and must perform a DMA transfer that spans physical pages—it must rely upon the actual contiguity of the physical pages involved in the transfer. Because there is no guarantee that this is the state of the user's buffer, the driver must allocate an intermediate buffer consisting of contiguous physical pages. The driver never deallocates this buffer unless the driver is being unloaded by means of SYSGEN's RELOAD command. (The driver unloading routine can call COM$DRVDEALMEM to do so.) The best time to allocate such a buffer is during the device's initialization, when memory is most likely to be contiguous.

The VMS routine EXE$ALOPHYCNTG, described in the *VMS Device Support Reference Manual*, allocates such a buffer. The size of the buffer that should be allocated depends on the device's characteristics and the size of the transfers requested on the device. A buffer of four pages is likely to be large enough for most disk transfers, for example; but if you have enough memory on your system, you might want to make your buffer the size of a disk track in order to reduce disk latency. In any event, large transfers to the device can be segmented into transfers the size of your intermediate buffer.

The start-I/O routine of such a driver copies the data from the user's buffer into the intermediate, physically contiguous buffer by means of the routine IOC$MOVFRUSER.

The driver then sets up the device for the DMA transfer:

1 Determines the physical address of the buffer from the system virtual address returned by EXE$ALOPHYCNTG

2 Moves the address to the device address register

3 Activates the device

4 If the transfer size exceeds the size of the buffer, returns to step 1

When a user requests a transfer from such a device, the driver moves the data from the device to the intermediate, physically contiguous buffer by means of a DMA transfer, then calls IOC$MOVTOUSER to copy the data into the user's buffer.

16.7 Unit Initialization Routine

A generic VAXBI device driver may include a unit initialization routine, in addition to its controller initialization routine, if it services a multiunit device.

SYSGEN attempts to create a UCB and call the unit initialization routine for the number of units specified in the **maxunits** argument to the DPTAB macro.

When called in the process of driver loading, the unit initialization routine of a generic VAXBI device driver must therefore determine if the unit it is currently servicing actually exists. Prior to returning control to SYSGEN, the routine must place in R0 a success status (low bit set) if the unit exists or a failure status (low bit clear) if it does not. If SYSGEN receives failure status, it deallocates the UCB for the unit and proceeds to configure the next unit in a similar manner.

16.8 Register Dumping Routine

In the event of a device error or a VAXBI bus error, a driver's register dumping routine should contain code that makes certain interesting registers available for error logging. Apart from any device registers that should be saved, the following BIIC registers may contain information important in determining the cause of the error: the Device Register (BIIC$L_DTREG), the VAXBI Control and Status Register (BIIC$L_BICSR), the Bus Error Register (BIIC$L_BER), the Error Interrupt Control Register (BIIC$L_EICR), and the Interrupt Destination Register (BIIC$L_IDR).

The following is an example of part of a register dumping routine that saves the contents of these BIIC registers in an error buffer.

```
MOVL    BIIC$L_DTREG(R4),(R0)+    ;Device Type Register
MOVL    BIIC$L_BICSR(R4),(R0)+    ;BIIC CSR Register
MOVL    BIIC$L_BER(R4),(R0)+      ;Bus Error Register
MOVL    BIIC$L_EICR(R4),(R0)+     ;Error Interrupt Control Register
MOVL    BIIC$L_IDR(R4),(R0)+      ;Interrupt Destination Register
```

16.9 Loading a VAXBI Device Driver

The System Generation Utility (SYSGEN) loads the device driver into system virtual memory, creates additional data structures for the device unit, connects the device's interrupt vectors, and calls the device driver's controller initialization routine and unit initialization routine.

Chapter 12 discusses the loading of a device driver and the SYSGEN commands commonly used during driver loading.

16.10 BIIC Register Definitions

Each VAXBI node is required to implement a minimum set of registers contained in specific locations within the node's node space. VMS automatically maps each node's node space at boot time and provides the macro $BIICDEF (in SYS$LIBRARY:LIB.MLB) to define offsets to the BIIC registers and their significant bit fields.

The contents of the BIIC registers are illustrated in Figure 16–6 and described in Table 16–1. See the *VAXBI Options Handbook* for a discussion of the BIIC and the rules for configuring its registers.

Note: Fields marked "Reserved to Digital" are reserved for Digital's future use and should contain zeros.

Figure 16–6 Backplane Interconnect Interface Chip (BIIC) Registers

BIIC$L_DTREG	0
BIIC$L_BICSR	4
BIIC$L_BER	8
BIIC$L_EICR	12
BIIC$L_IDR	16
BIIC$L_IPIMR	20
BIIC$L_IPIDR	24
BIIC$L_IPISR	28

(continued on next page)

Figure 16–6 (Cont.) Backplane Interconnect Interface Chip (BIIC) Registers

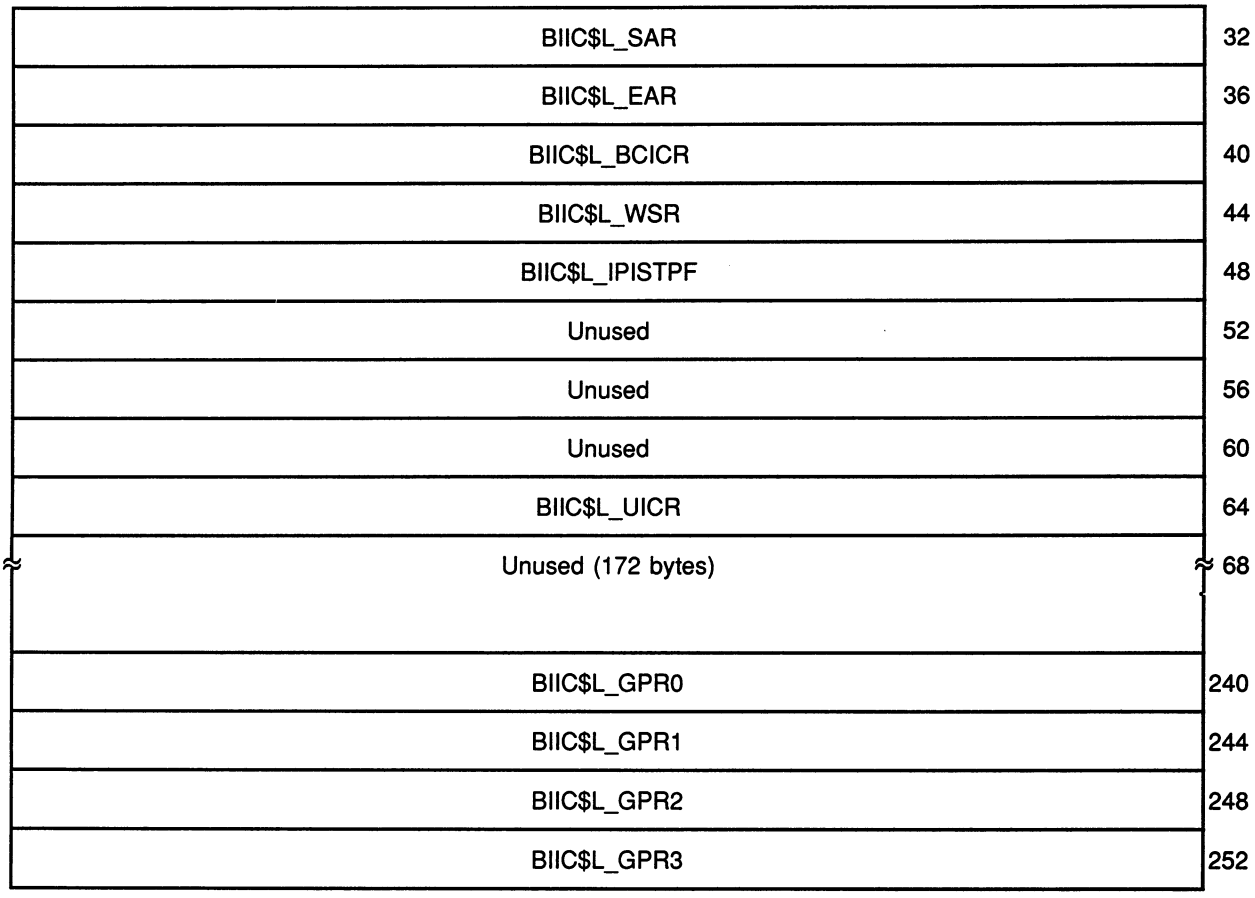


Table 16–1 Contents of the BIIC Registers

Field Name	Contents	
BIIC$L_DTREG	Device Register that contains the following two words:	
	BIIC$W_DEVTYPE	Device type. This field is written by device hardware and self-test microcode. It contains two bit fields: BIIC$V_MEMNODE (bits <14:8>), when clear, indicates a memory node. BIIC$V_NONDEC (bit 15), when clear, indicates a Digital-supplied device; it should be 1 otherwise.
	BIIC$W_REVCODE	Revision code.

(continued on next page)

Table 16–1 (Cont.) Contents of the BIIC Registers

Field Name	Contents	
BIIC$L_BICSR	VAXBI Control and Status Register.	
	The following fields are defined within BIIC$L_BICSR.	
	BIIC$V_NODE_ID[1]	Node ID. This field is automatically loaded during the powerup sequence. Reserved to Digital.
	BIIC$V_ARBCNTL	Arbitration mode used by the node. Currently, all arbitration modes except dual round-robin arbitration are reserved to Digital. Correspondingly, these two bits should be clear. When these two bits are set, arbitration is disabled, thus preventing a node from starting a VAXBI transaction.
	BIIC$V_SEIE	Soft error interrupt enable. When set, this bit allows the node to generate an interrupt when the soft error summary bit (BIIC$V_SES) in this register is set.
	BIIC$V_HEIE	Hard error interrupt enable. When set, this bit allows the node to generate an interrupt when the hard error summary bit (BIIC$V_HES) in this register is set.
	BIIC$V_UWP	Unlock write pending. When set, this bit signals that the master port interface at this node has successfully completed an IRCI (Interlock Read with Cache Intent) transaction. The node clears this bit when it successfully completes a corresponding UWMCI (Unlock Write Mask with Cache Intent) instruction.
	<9>	Reserved to Digital. Must be zero.
	BIIC$V_SST	Node reset. This bit is normally used by diagnostics to initiate the BIIC internal self-test. Prior to initiating a BIIC self-test, a node should disable arbitration by setting both bits in BIIC$V_ARBCNTL. When BIIC$V_SST is set, the self-test status bit (BIIC$V_STS) in this register must also be set. Reads to BIIC$V_SST return a zero.
	BIIC$V_STS	Self-test status. When set, this bit indicates that the BIIC has passed its self-test. The controller initialization routine of a VAXBI device driver should inspect this bit and the BIIC$V_BROKE bit before proceeding with any VAXBI transactions. During the self-test sequence, BIIC$V_STS will automatically be reset by the BIIC to allow the proper recording of the new self-test results at the end of self-test.
	BIIC$V_BROKE[2]	Broke bit. When cleared by the device's self-test, this bit indicates that the device has passed its self-test. The controller initialization routine of a VAXBI device driver should inspect this bit and the BIIC$V_STS bit before proceeding with any VAXBI transactions.
	BIIC$V_INIT[2]	Initialization bit.

[1] Read-only field.

[2] Write-one-to-clear bit. Write-type transactions cannot set this bit.

(continued on next page)

Table 16–1 (Cont.) Contents of the BIIC Registers

Field Name	Contents	
	BIIC$V_SES[1]	Soft error summary. When set, this bit indicates that one or more of the soft error bits in the Bus Error Register (BIIC$L_BER) is set.
	BIIC$V_HES[1]	Hard error summary. When set, indicates that one or more of the hard error bits in the Bus Error Register (BIIC$L_BER) is set.
	BIIC$V_BIICTYPE[1]	BIIC type. These bits <23:16> always contain 00000001.
	BIIC$V_BIICREVN[1]	BIIC revision number.
BIIC$L_BER	Bus Error Register.	
	The following bits are defined within BIIC$L_BER. Bits <30:16> are hard error bits and bits <2:0> are soft error bits.	
	BIIC$V_NPE[2]	Null bus parity error.
	BIIC$V_CRD[2]	Corrected read data.
	BIIC$V_IPE[2]	ID parity error.
	BIIC$V_UPEN[1]	User parity enabled.
	<14:4>[1]	Reserved to Digital. Must be zero.
	BIIC$V_ICE[2]	Illegal confirmation error.
	BIIC$V_NEX[2]	Nonexistent address.
	BIIC$V_BTO[2]	Bus timeout.
	BIIC$V_STO[2]	Stall timeout.
	BIIC$V_RTO[2]	Retry timeout.
	BIIC$V_RDS[2]	Read data substitute.
	BIIC$V_SPE[2]	Slave parity error.
	BIIC$V_CPE[2]	Command parity error.
	BIIC$V_IVE[2]	IDENT vector error.
	BIIC$V_TDF[2]	Transmitter during fault.
	BIIC$V_ISE[2]	Interlock sequence error.
	BIIC$V_MPE[2]	Master parity error.
	BIIC$V_CTE[2]	Control transmit error.
	BIIC$V_MTCE[2]	Master transmit check error.
	BIIC$V_NMR[2]	NO ACK to multiresponder command received.
	<31>	Reserved to Digital. Must be zero.

[1]Read-only field.

[2]Write-one-to-clear bit. Write-type transactions cannot set this bit.

(continued on next page)

Table 16–1 (Cont.) Contents of the BIIC Registers

Field Name	Contents
BIIC$L_EICR	Error Interrupt Control Register. This register supplies information the node uses to request and monitor the status of both BIIC-detected and forced-error interrupts: that is, those interrupts signaled by either the setting of a bit in the Bus Error Register (BIIC$L_BER) or the setting of the force bit (BIIC$V_EIFORCE) in this register, respectively. The node can initiate BIIC-detected error-interrupt requests only if the appropriate error-interrupt enables (BIIC$V_SEIE and/or BIIC$V_HEIE) are set in the VAXBI Control and Status Register (BIIC$L_BICSR). The following fields are defined within BIIC$L_EICR. <1:0>[1] Reserved to Digital. Must be zero. BIIC$V_EIVECTOR 12-bit vector used in error interrupt sequences. <15:14>[1] Reserved to Digital. Must be zero. BIIC$V_LEVEL These four bits (<19:16>) correspond to the four interrupt levels (INT<7:4>) of the VAXBI bus. A set bit causes the corresponding level to be used when INTR commands under control of this register are transmitted. BIIC$V_EIFORCE Force bit. When set, this bit posts an error interrupt request in the same way as a bit set in the Bus Error Register (BIIC$L_BER), except that the request is not qualified by the bits BIIC$V_HEIE and BIIC$V_SEIE in BIIC$L_BICSR. BIIC$V_EISENT[2] INTR sent. <22> Reserved to Digital. Must be zero. BIIC$V_EIINTC[2] INTR complete. When set, this bit indicates that the vector for an error interrupt has been successfully transmitted or an INTR command sent under the control of this register has been successfully aborted. BIIC$V_EIINTAB[2] INTR abort. When set, this bit indicates that an INTR command under the control of this register has been aborted (that is, a NO ACK or illegal confirmation code has been received). This bit is a status bit set by the BIIC and can be reset only by the user interface. <31:25>[1] Reserved to Digital. Must be zero.
BIIC$L_IDR	Interrupt Destination Register. The low-order word of this register indicates which nodes are to be selected by INTR commands.
BIIC$L_IPIMR	Interprocessor Interrupt Mask Register. The high-order word of this register indicates which nodes are permitted to send interprocessor interrupts to this node.
BIIC$L_IPIDR	Force-bit IPINTR/STOP Destination Register. The low-order word of this register indicates which nodes are to be targeted by force-bit IPINTR or STOP commands sent by this node.
BIIC$L_IPISR[2]	IPINTR Source Register. The BIIC stores in the high-order word of this register the decoded ID of a node that sends an IPINTR command to this node.

[1]Read-only field.

[2]Write-one-to-clear bit. Write-type transactions cannot set this bit.

(continued on next page)

Table 16–1 (Cont.) Contents of the BIIC Registers

Field Name	Contents
BIIC$L_SAR	Starting Address Register. The Starting Address Register and Ending Address Register define storage blocks in either memory or I/O space. They must not be configured to include node space or multicast space. The low-order 18 bits of this register must be zero. This means that memories are multiples of 256K bytes. Software should set up the Starting Address Register before the Ending Address Register.
BIIC$L_EAR	Ending Address Register. The low-order 18 bits of this register must be zero. This means that memories are multiples of 256K bytes. Software should set up the Starting Address Register before the Ending Address Register. See the description of the Starting Address Register (BIIC$L_SAR) for further details..
BIIC$L_BCICR	BCI Control Register. The following fields are defined within BIIC$L_BCICR. <2:0>[1] Reserved to Digital. Must be zero. BIIC$V_RTOEVEN RTO EV enable. BIIC$V_PNXTEN Pipeline NXT enable. BIIC$V_IPINTREN IPINTR enable. BIIC$V_INTREN INTR enable. BIIC$V_BICSREN BIIC CSR Space enable. BIIC$V_UCSREN User Interface CSR Space enable. BIIC$V_WINVALEN WRITE Invalidate enable. BIIC$V_INVALEN INVAL enable. BIIC$V_IDENT IDENT enable. BIIC$V_RESEN RESERVED enable. BIIC$V_STOPEN STOP enable. BIIC$V_BDCSTEN BDCST enable. BIIC$V_MSEN Multicast Space enable. BIIC$V_IPINTRF IPINTR/STOP force. BIIC$V_BURSTEN Burst enable. <31:18>[1] Reserved to Digital. Must be zero.
BIIC$L_WSR	Write Status Register. The following fields are defined within BIIC$L_WSR. <27:0>[1] Reserved to Digital. Must be zero. BIIC$V_GPR0[2] Indicates that a VAXBI transaction has written to General Purpose Register 0 (BIIC$L_GPR0).

[1] Read-only field.

[2] Write-one-to-clear bit. Write-type transactions cannot set this bit.

(continued on next page)

Table 16–1 (Cont.) Contents of the BIIC Registers

Field Name	Contents	
	BIIC$V_GPR1[2]	Indicates that a VAXBI transaction has written to General Purpose Register 1 (BIIC$L_GPR1).
	BIIC$V_GPR2[2]	Indicates that a VAXBI transaction has written to General Purpose Register 2 (BIIC$L_GPR2).
	BIIC$V_GPR3[2]	Indicates that a VAXBI transaction has written to General Purpose Register 3 (BIIC$L_GPR3).
BIIC$L_IPISTPF	Force-Bit IPINTR/STOP Command Register.	
	The following fields are defined within BIIC$L_IPISTPF.	
	<10:0>[1]	Reserved to Digital. Must be zero.
	BIIC$V_MIDEN	Master ID Enable.
	BIIC$V_CMD	These four bits indicate the command code for either an IPINTR or STOP transaction that is initiated by setting the IPINTR/STOP force bit (BIIC$V_INTRF in BIIC$L_BCICR).
	<31:16>[1]	Reserved to Digital. Must be zero.
BIIC$L_UICR	User Interface Interrupt Control Register. This register controls the operation of interrupts initiated by the device.	
	The following fields are defined within BIIC$L_UICR.	
	<1:0>[1]	Reserved to Digital. Must be zero.
	BIIC$V_UIVECTOR	These 12 bits contain the vector used during user interface interrupt sequences (unless the external vector bit (BIIC$V_EXVECTOR in BIIC$L_UICR) is set). The vector is transmitted when this node wins an IDENT arbitration that matches the conditions given in BIIC$L_UICR.
	<14>	Reserved to Digital. Must be zero.
	BIIC$V_EXVECTOR	When set, the BIIC solicits the interrupt vector from the node rather than transmitting the vector contained in BIIC$L_UICR.
	BIIC$V_UIFORCE	These four bits correspond to the four interrupt levels (INT<7:4>). When a bit is set, the BIIC generates an interrupt at the indicated level.
	BIIC$V_UISENT[2]	These four bits correspond to the four interrupt levels (INT<7:4>). A set bit indicates that an INTR command for the corresponding level has been successfully transmitted.
	BIIC$V_UIINTC[2]	These four bits correspond to the four interrupt levels (INT<7:4>). A set bit indicates that the vector for an interrupt at the corresponding level has been successfully transmitted or that an INTR command sent under the control of this register has been successfully aborted.

[1] Read-only field.

[2] Write-one-to-clear bit. Write-type transactions cannot set this bit.

(continued on next page)

Table 16–1 (Cont.) Contents of the BIIC Registers

Field Name	Contents	
	BIIC$V_UIINTAB[2]	These four bits correspond to the four interrupt levels (INT<7:4>). A set bit indicates that an INTR command at the corresponding level, sent under the control of this register, has been aborted (that is, a NO ACK or illegal confirmation code has been received).
BIIC$L_GPR0	General Purpose Register 0.	
BIIC$L_GPR1	General Purpose Register 1.	
BIIC$L_GPR2	General Purpose Register 2.	
BIIC$L_GPR3	General Purpose Register 3.	

[2]Write-one-to-clear bit. Write-type transactions cannot set this bit.

17 SCSI Class Driver Support

The Small Computer System Interface (SCSI) provides a standard bus by which small computers and intelligent peripheral devices may be interconnected. The VMS operating system offers a native mode implementation of the ANSI SCSI bus on its MicroVAX/VAXstation 3100 and VAXstation 3520/3540 system configurations. Any non-Digital-supplied device to be attached to the SCSI bus of a MicroVAX/VAXstation system must implement all mandatory features of the SCSI–2 standard as described in the specification.

The VMS operating system defines a mechanism by which a system programmer can write a class driver that, in conjunction with a standard VMS SCSI port driver, exchanges data, commands, and status with a third-party device on the SCSI bus. Given the particular requirements of the device, or the expectations of application programs accessing the device, the programmer may choose to create a SCSI class driver rather than employ a VMS generic SCSI class driver dicussed in the *VMS I/O User's Reference Manual: Part I*.

By writing a device-specific SCSI class driver, a programmer can define a unique, simple, robust $QIO interface to a SCSI device. The generic SCSI class driver, by contrast, provides a more complex $QIO interface, requiring the application program to have some knowledge of the data transfer mode and capabilities of the target device and to construct in memory the SCSI commands to be passed to the SCSI port. A third-party SCSI class driver conceals these details from the application program. Additionally, it can provide device-specific error recovery, full error logging, and notification of asynchronous events from the device.

Note: A non-Digital-supplied SCSI disk device residing on the local node and controlled by a SCSI third-party class driver cannot be served to other nodes of the local area VAXcluster.

This chapter introduces the VMS SCSI class/port interface and discusses the mechanisms VMS provides to facilitate the creation of a SCSI class driver. It describes the capabilities and components of such a driver and suggests some coding strategies. It also includes sections on driver naming conventions, driver loading, and driver debugging techniques. It concludes with descriptions of class driver error logging protocol and the asynchronous event notification facility.

17.1 VAX Systems with SCSI Bus Concepts

Implementation of the SCSI bus architecture for the MicroVAX/VAXstation 3100 is shown in Figure 1–6 and for the VAXstation 3520/3540 in Figure 1–7. Each SCSI bus in the system is identified by a **SCSI port ID** (*A* or *B*). The SCSI port ID uniquely identifies a **SCSI port**: that is, the SCSI controller channel that controls communications to and from a specific SCSI bus on the system.

Each SCSI bus supports seven devices and a processor, at SCSI IDs 0 to 7. As defined by the ANSI SCSI specification, a **SCSI ID** refers to a line on the SCSI data bus (DB) on which the device uniquely asserts itself. The VMS operating system uses the term **SCSI device ID** to represent this value. Typically, a MicroVAX/VAXstation 3100 system processor is assigned device ID 6 and asserts itself at DB(6); a VAXstation 3520/3540 system processor is assigned device ID 7 and asserts itself at DB(7).

According to the ANSI SCSI specification, a **logical unit** is a physical or virtual device accessible by means of a SCSI device. For instance, if a peripheral controller resides on the SCSI bus, it, in turn, can control up to eight devices. A **logical unit number** (LUN), an integer from 0 to 7, uniquely identifies the device with respect to the controller's SCSI device ID.

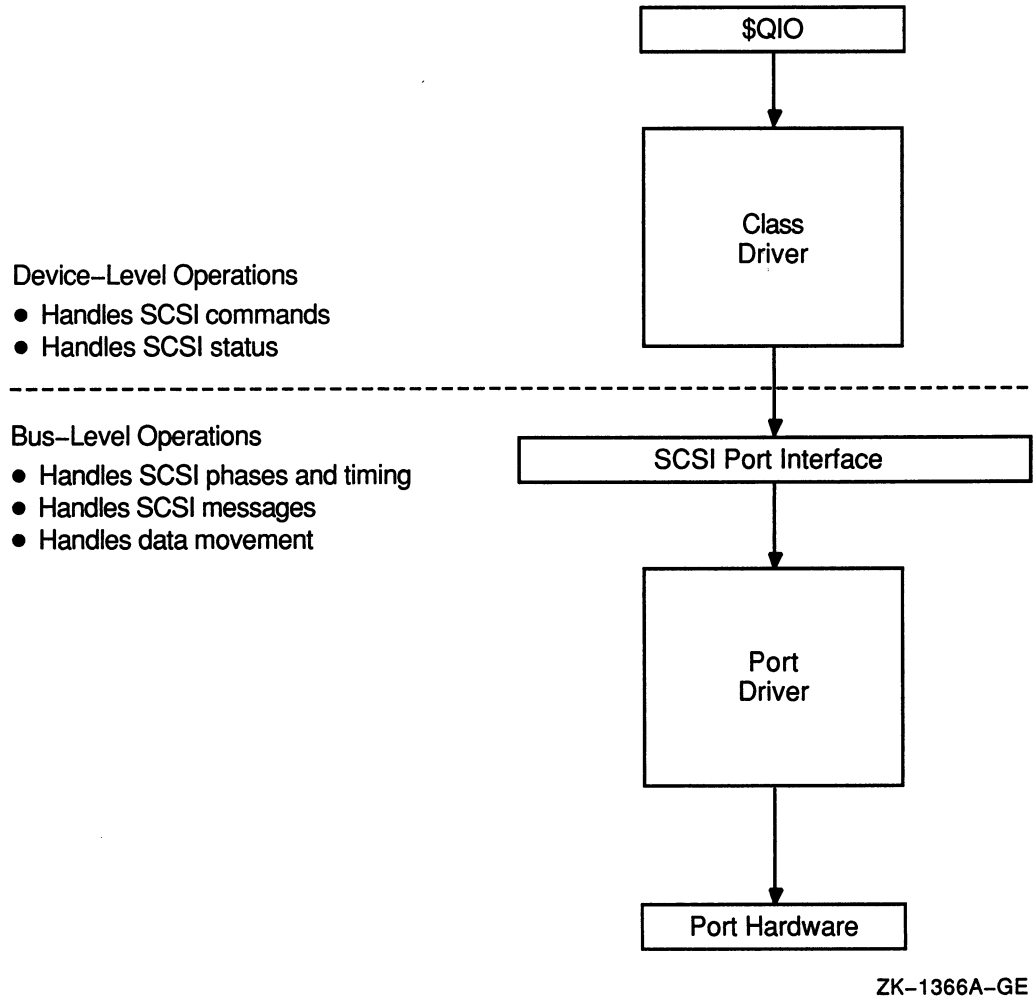
Transactions on the SCSI bus are between an **initiator** and a **target**. The initiator, usually the host processor, requests that another SCSI device, the target, perform a certain operation. In situations in which the host processor requires notification of some unexpected event on the SCSI bus, the ANSI specification defines the **asynchronous event notification** (AEN) protocol. AEN allows a SCSI device that is usually a target to inform the processor that an event has occurred asynchronously with respect to the processor's current stream of execution. (Certain MicroVAX/VAXstation implementations make the AEN protocol available to non-Digital-supplied SCSI class drivers, as described in Section 17.9.)

As Figures 1–6 and 1–7 illustrate, the MicroVAX/VAXstation 3100 and VAXstation 3520/3540 port hardware cannot directly access data in main memory. In order to access command, status, and data buffers involved in an operation on the SCSI bus, the MicroVAX/VAXstation port hardware must refer to its own direct-memory-access (DMA) buffer. Whenever the port hardware requires access to buffered information, the standard VMS port driver dynamically allocates a segment of the **port DMA buffer** and maps to it the pages of the buffer in main memory in a system-dependent manner.

17.2 SCSI Class/Port Architecture

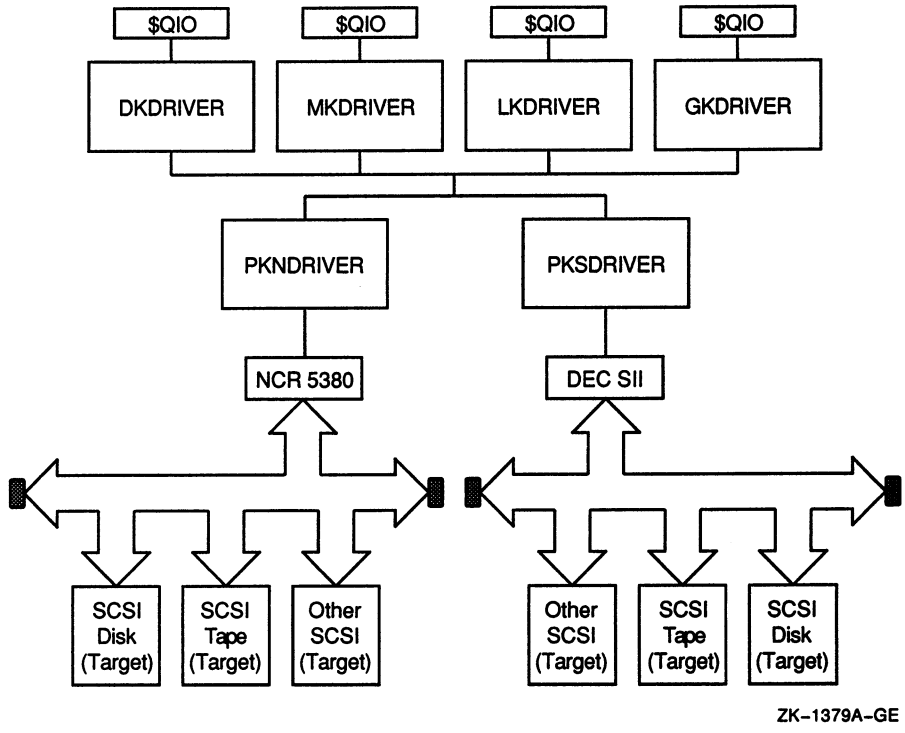
VMS uses a class/port driver architecture to communicate with devices on the SCSI bus. The class/port design allows the responsibilities for communication between the operating system and the device to be cleanly divided between two separate driver images (see Figure 17–1).

Figure 17–1 VMS SCSI Class/Port Interface



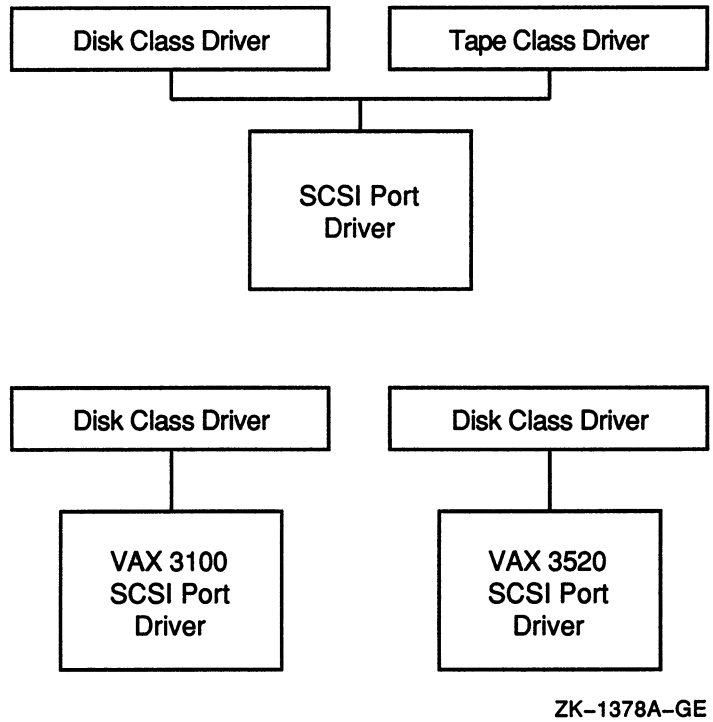
The **SCSI port driver** transmits and receives SCSI commands and data. It knows the details of transmitting data from the local processor's SCSI port hardware across the SCSI bus. Although it understands SCSI bus phases, protocol, and timing, the SCSI port driver has no knowledge of the SCSI commands the device supports, the status messages it returns, or the format of the packets in which this information is delivered. Strictly speaking, the port driver is a communications path. When directed by a SCSI class driver, the port driver forwards commands and data from the class driver onto the SCSI bus to the device. On a single MicroVAX/VAXstation system, a single SCSI port driver handles bus-level communications for all SCSI class drivers that may exist on the system (see Figure 17–2).

Figure 17–2 VMS SCSI Port Driver Configuration



The **SCSI class driver** acts as an interface between the user and the SCSI port, translating I/O functions as specified in a user's $QIO request to a SCSI command directed to a device on the SCSI bus. Although the class driver knows about SCSI command descriptor buffers, status codes, and data, it has no knowledge of underlying bus protocols or hardware, command transmission, bus phases, timing, or messages (except in asynchronous event notification mode, as described in Section 17.9). A single SCSI class driver can run with the SCSI port driver of any MicroVAX/VAXstation system, controlling the same set of devices on each system (see Figure 17–3).

Figure 17–3 VMS SCSI Class Driver Configuration



The design of the SCSI driver class/port interface allows a programmer to write a class driver that is independent of any concern about the underlying hardware. VMS supplies software tools that facilitate the development of SCSI class drivers, including the following:

- A standard interface that all SCSI class drivers use to request work from and transfer control to the port driver. This interface is known as the **SCSI port interface** (SPI).
- SCSI-specific data structures that class and port drivers use to exchange information and monitor the state of the device connection or SCSI port.
- A template SCSI class driver that can serve as the basis for a third-party SCSI class driver.

17.2.1 SCSI Port Interface

The SCSI port interface (SPI) consists of a group of routines within the SCSI port driver that create and manage the connection between a SCSI class driver and a device unit. Across this connection, SPI routines exchange control information and data between the class driver and the port.

When a connection must be established, a SCSI command transmitted, or data transferred, a SCSI class driver calls the appropriate routine within the port driver by invoking one of a series of macros, defined in SYS$LIBRARY:LIB.MLB. Each macro corresponds to a vector in the SCSI port descriptor table (SPDT) that supplies the address of the port routine that performs the applicable function. Table 17–1 lists the standard SPI macros and their functions.

Table 17–1 SCSI Port Interface (SPI) Macros

Macro	Description
SPI$ABORT_COMMAND	Aborts the execution of an outstanding SCSI command over a specified connection
SPI$ALLOCATE_COMMAND_BUFFER	Allocates a buffer in which a class driver passes a SCSI command descriptor to the port driver
SPI$CONNECT	Creates a connection from a class driver to a SCSI device unit
SPI$DEALLOCATE_COMMAND_BUFFER	Deallocates a SCSI command buffer
SPI$DISCONNECT	Breaks the connection between a class driver and a SCSI device unit
SPI$GET_CONNECTION_CHAR	Obtains the characteristics of a specified connection and places them in the buffer specified by the class driver
SPI$MAP_BUFFER	Makes the process buffer involved in a data transfer available to the port driver
SPI$RESET	Resets the port hardware and SCSI bus
SPI$SEND_COMMAND	Delivers a SCSI command descriptor buffer to a SCSI device, returning status and data, if applicable
SPI$SET_CONNECTION_CHAR	Sets up the characteristics of a specified connection
SPI$UNMAP_BUFFER	Releases the SCSI port's DMA buffer space and the system page-table entries that double-mapped a user buffer involved in a transfer

A SCSI class driver invokes SPI macros at fork IPL, holding the fork lock. Because the port driver routines called by SPI macros may fork or stall, a class driver must preserve local context and local return addresses across an SPI macro invocation. It must also ensure that the address of its caller is at the top of the stack at the time the macro is invoked. (These issues are more fully discussed in Section 17.6.1.)

Detailed descriptions of the functions provided by the SPI macros appear where pertinent in the discussions of SCSI class driver operations that follow in this chapter. The macro chapter of the *VMS Device Support Reference Manual* provides a condensed description of the calling interface, functions, inputs, and returned values of each macro.

An extension to the SPI interface includes several additional macros that enable the host to receive an asynchronous event notification from a target on the SCSI bus. Section 17.9 describes the asynchronous event notification (AEN) feature in greater detail, and introduces each of the macros in the SPI interface extension.

17.2.2 SCSI-Specific Data Structures

The SCSI class/port interface must maintain status and control information relevant to each participating connection and port. Moreover, SCSI class drivers and port drivers require a means of sharing information about each I/O request that involves the port. The following data structures accommodate these needs:

- SCSI connection descriptor table (SCDT)
- SCSI port descriptor table (SPDT)
- SCSI class driver request packet (SCDRP)
- Device and port unit control blocks (UCBs)

The **SCSI connection descriptor table** (SCDT) contains information specific to a connection established between a SCSI class driver and the port, such as phase records, timeout values, and error counters. The SCSI port driver creates an SCDT each time a SCSI class driver, by invoking the SPI$CONNECT macro, connects to a device on the SCSI bus. The class driver stores the address of the SCDT in the SCSI device's UCB.

The port driver has exclusive access to the SCDT; it is not accessed by the class driver. (The SCDT structure is shown and described in the data structure chapter of the *VMS Device Support Reference Manual.*)

The **SCSI port descriptor table** (SPDT) contains information specific to a SCSI port, such as the port driver connection database. The SPDT also includes a set of vectors, corresponding to the SPI macros invoked by SCSI class drivers, that point to service routines within the port driver. The SCSI port driver's unit initialization routine creates an SPDT for each SCSI port defined for a specific MicroVAX/VAXstation system and initializes each SPI vector.

The port driver reads and writes fields in the SPDT. The class driver does not write SPDT fields, but reads the SPDT indirectly when it invokes an SPI macro. (The SPDT structure is shown and described in the *VMS Device Support Reference Manual.*)

A SCSI class driver creates a **SCSI class driver request packet** (SCDRP) to deliver to the port driver information specific to an I/O request, such as the address of the SCSI command descriptor buffer. The class driver also places in the SCDRP some of the data it originally received in the I/O-request packet (IRP), such as the $QIO system service parameters, the I/O function, and the length and location of any user-specified buffer involved in a transfer. The port driver returns the actual data transfer byte count and status information to the class driver in the SCDRP.

Both class and port drivers read and write fields in the SCDRP; the port driver may modify fields written by the class driver. (The SCDRP structure is shown and described in the *VMS Device Support Reference Manual.*)

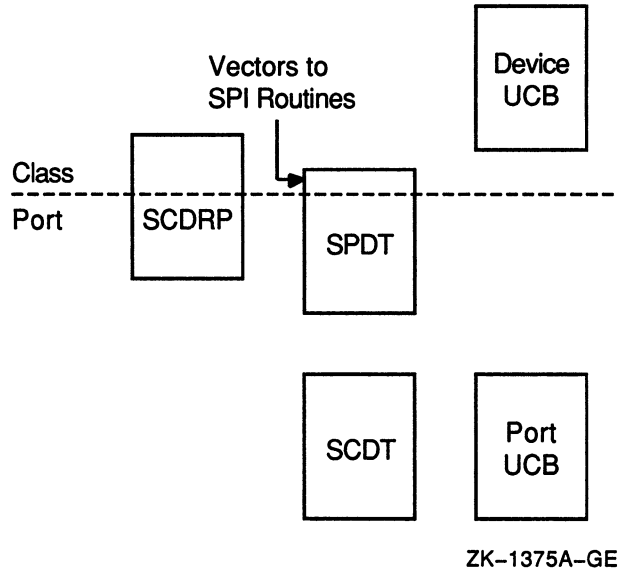
Two unit control blocks (UCBs) are involved in any interaction between the class driver and the port. The SCSI class driver maintains information in the **SCSI device UCB**, such as the device type, class, and characteristics; maximum transfer size; the address of the current SCDRP; and the addresses of the associated SPDT and SCDT. The SCSI port driver maintains similar information in the **SCSI port UCB**.

Table 17–2 summarizes the class/port ownership of and access to these structures; their interrelationships are pictured in Figure 17–4.

Table 17–2 Data Structures

Structure	Allocation	Owner	Port Access	Class Access
SCDRP	One per I/O transfer request	Class driver	Read/write	Read/write
SCDT	One per SCSI connection	Port driver	Read/write	None
SPDT	One per SCSI port	Port driver	Read/write	Read
SCSI device UCB	One per SCSI device unit	Class driver	None	Read/write
SCSI port UCB	One per SCSI controller port	Port driver	Read/write	None

Figure 17–4 SCSI Class/Port Data Structures



17.2.3 SCSI Class Driver Template

The VMS operating system supplies a model for a third-party class driver in the SCSI class driver template, located in SYS$EXAMPLES:SKDRIVER.MAR and listed in Appendix B.

SKDRIVER is a simplified, self-documenting driver that supports the I/O functions IO$_AVAILABLE, IO$_DIAGNOSE, and IO$_READLBLK on a generalized device. SKDRIVER performs most operations required of a typical SCSI class driver to process a typical I/O request, including the appropriate SPI interface macro calls to establish a connection to the port, allocate port resources, and accomplish a transfer to the SCSI device. SKDRIVER also allocates pool for two SCDRPs. It uses one to send a SCSI command to the device, and it uses the other to issue a SCSI REQUEST SENSE command in the event the SCSI device returns failure status on the original command (see Section 17.5.6.2 for information on the interpretation of port and SCSI status return values).

In addition, SKDRIVER defines local macros that simplify common operations, including the following:

- Preserving register contents and return addresses within the class driver across calls to executive and port routines that may destroy this context (INIT_UCB_STACK, SUBPOP, SUBPUSH, SUBRETURN).
- Assembling the information relevant to a supported SCSI command such that a driver routine can easily construct the SCSI command descriptor buffer and initialize the SCDRP fields describing transfer buffer characteristics and timeout values (SCSI_CMD). SKDRIVER uses this macro to define the SCSI commands TEST UNIT READY, INQUIRY, REQUEST SENSE, and MODE SELECT. (It "invents" a fifth command, QIO INQUIRY, to provide a device-independent read operation servicing an IO$_READLBLK I/O function.)

SKDRIVER extends the device UCB to accommodate its context-saving macros, the allocation of SCDRPs, and per-request timeout values (see Section 17.4.2), SCSI-specific device characteristics, and the addresses of the SCDT and the current SCDRP.

Code in the SCSI class driver template can serve as a good starting point for the development of a third-party SCSI class driver. Subsequent sections of this chapter refer to the SCSI class driver template, as appropriate, to explain certain class driver concepts or possible implementation strategies.

17.3 Connecting to a SCSI Device

As defined by the VMS SCSI class/port interface, a **connection** is a logical link between a SCSI class driver and a SCSI device unit. In MicroVAX/VAXstation systems, a SCSI device is identified by its device mnemonic (for instance, *SK*), its SCSI port ID (*A* or *B*), its SCSI device ID (an integer from 0 to 7), and its logical unit number (an integer from 0 to 7).

Before a SCSI class driver can issue commands to a target device on the SCSI bus and transfer data across the bus, it must establish a logical connection to that device. The SPI$CONNECT macro connects a SCSI class driver with a target device, thereby establishing a linkage between the SCSI class driver and the SCSI port driver. Once the SCSI connection exists, the class and port drivers can intercommunicate.

A SCSI class driver's unit initialization routine invokes the SPI$CONNECT macro at fork level, specifying the SCSI port ID (in numeric form), the SCSI device ID, and the SCSI logical unit number of the device to which it needs to connect. (More detailed information about the use and functions of the SPI$CONNECT macro appears in Section 17.8.6 and the *VMS Device Support Reference Manual.*)

Normally a connection lasts throughout the runtime life of a system; a SCSI class driver should never need to break a connection.

17.4 Setting Up a SCSI Command

This section describes the procedures a SCSI class driver follows to set up a SCSI command for transmission to the SCSI port driver. Although it discusses the aspects of the setup of a data transfer over the SCSI bus that relate to the preparation of a SCSI command, you should refer to Section 17.5 for a more complete discussion of SCSI data transfers.

17.4.1 Preparing a SCSI Command Descriptor Block

In preparation for sending a SCSI command to a device on the SCSI bus, a SCSI class driver must first determine which SCSI commands it supports. For each supported SCSI command, the driver programmer must perform the following tasks:

- Determine the correct size and format for the command
- Define the appropriate contents for all command bytes
- Allocate a SCSI port command buffer to make the command descriptor block and status buffer available to the port
- Create a SCSI command descriptor block in the SCSI port command buffer
- Create a 1-byte SCSI status buffer in the SCSI port command buffer
- Establish pointers in the SCDRP to the command descriptor block and the status buffer
- If the command involves a data transfer, store the parameters of the transfer in the SCDRP

The SCSI class driver template (SKDRIVER) performs these operations by means of the locally-defined SCSI_CMD macro and the SETUP_CMD subroutine. Each invocation of the SCSI_CMD macro establishes a data area within the driver to contain information about a specific SCSI command, including its length and the contents of its command bytes, plus the size, direction, and timeout values for any associated data transfer. The SCSI class driver template uses the SCSI_CMD macro to define

the parameters of five 6-byte SCSI commands, although the macro can describe commands of any length.

The SETUP_CMD subroutine of the class driver template SK_STARTIO routine repackages command data into a SCSI command descriptor block. Because both the command descriptor block and the SCSI status buffer must be accessed by both the class and port drivers, it is useful to account for the status buffer in the request to allocate the SCSI port command buffer. Thus, the SETUP_CMD subroutine adds 2 longwords of overhead—one for the SCSI status byte and one for the SCSI command size—to the SCSI command size. It then invokes SPI$ALLOCATE_COMMAND_BUFFER, causing the port driver to allocate a port command buffer and return its address and size.

The class driver initializes the status longword to –1 and stores its address in SCDRP$L_STS_PTR. It places the size (in bytes) of the SCSI command in the next longword, and then constructs a SCSI command descriptor block in the buffer, copying the command to the buffer byte by byte. It places the address of the size longword in SCDRP$L_CMD_PTR.

Prior to invoking SPI$SEND_COMMAND to transmit the command descriptor block to the port driver, the class driver may perform several optional tasks to set up a data transfer operation, such as the following:

- Initializing the phase change (DMA) timeout and/or disconnect timeout fields in the SCDRP (SCDRP$L_DMA_TIMEOUT and SCDRP$L_DISCON_TIMEOUT), thus providing command-specific timeout values (see Section 17.4.2 for information on how to set up timeout values)
- For a data transfer involving a user buffer, initializing fields in the SCDRP to reflect the parameters of the buffer, and acquiring a port mapping of that buffer
- For a data transfer requiring a system buffer, allocating the buffer from nonpaged pool, initializing fields in the SCDRP to reflect the parameters of the buffer, and acquiring a port mapping of that buffer

When the command has completed and the SCSI port command buffer is no longer required, the class driver checks the command status, as described in Section 17.5.6, and invokes the SPI$DEALLOCATE_COMMAND_BUFFER macro to deallocate the buffer.

17.4.2 Setting Command Timeouts

The SCSI port driver implements several timeout mechanisms, some governed by the ANSI SCSI specification and others required by VMS. The timeouts required by VMS include the following:

Timeout	Description
Phase change timeout	Maximum number of seconds for a target to change the SCSI bus phase or complete a data transfer. (This value is also known as the **DMA timeout**.) Upon sending the last command byte, the port driver waits this many seconds for the target to change the bus phase lines and assert REQ (indicating a new phase). Or, if the target enters the DATA IN or DATA OUT phase, the transfer must be completed within this interval.
Disconnect timeout	Maximum number of seconds, from the time the initiator receives the DISCONNECT message, for a target to reselect the initiator so that it can proceed with the disconnected I/O transfer.

The SCSI class driver is responsible for maintaining both of these timeout values. It has the following three options:

- Accepting a connection's default value. The default value for both timeouts is 4 seconds.
- Altering the connection's default value. To modify the default values, the class driver specifies nonzero values in the **phase change timeout** and **disconnect timeout** longwords of the connection characteristics buffer and invokes the SPI$SET_CONNECTION_CHAR macro.
- Establishing timeouts for individual commands that override the connection's default value. If, prior to invoking the SPI$SEND_COMMAND macro, the class driver supplies a nonzero value in either SCDRP$L_DMA_TIMEOUT or SCDRP$L_DISCON_TIMEOUT, the port driver uses that value, instead of the default, for the course of that data transfer.

17.4.3 Disabling Command Retry

The SCSI port driver implements a command retry mechanism, which is enabled on a given connection by default.

When the command retry mechanism is enabled, the port driver retries up to three times any I/O operation that fails during the COMMAND, Message, Data, or STATUS phases. For instance, if the port driver detects a parity error during the Data phase, it aborts the I/O operation, logs an error, and retries the I/O operation. It repeats this sequence twice more, if necessary. If the I/O operation completes successfully during a retry attempt, the port driver returns success status to the class driver. However, if all retry attempts fail, the port driver returns failure status to the class driver.

When command retry is enabled on a connection, a SCSI class driver can control the number of retries the port attempts by supplying nonzero values in the **command retry count**, **busy retry count**, **arbitration retry count**, and **select retry count** longwords of the connection characteristics buffer, and invoking the SPI$SET_CONNECTION_CHAR macro.

A SCSI class driver may need to disable the command retry mechanism under certain circumstances. For instance, repeated execution of a command on a sequential device may produce different results than are intended by a single command request. A tape drive could perform a partial write and then repeat the write without resetting the tape position.

A SCSI class driver can disable this mechanism by setting bit 1 of the **connection flags** longword of the connection characteristics buffer, and invoking the SPI$SET_CONNECTION_CHAR macro.

17.5 Performing a SCSI Data Transfer

This section describes the procedures a SCSI class driver follows to set up and accomplish a data transfer over the SCSI bus.

17.5.1 Setting the Data Transfer Mode

The SCSI bus defines two data transfer modes, asynchronous and synchronous. In asynchronous mode, for each REQ from a target there is an ACK from the host prior to the next REQ from the target. Synchronous mode allows higher data transfer rates by allowing a pipelined data transfer mechanism where, for short bursts (defined by the REQ-ACK offset), the target can pipeline data to an initiator without waiting for the initiator to respond.

A class driver can determine the transfer modes supported by a device from the port capabilities longword returned from its invocation of the SPI$CONNECT macro. Whether or not a port or a target device supports synchronous data transfers, it is harmless for a class driver to set up the connection to use such transfers. If synchronous mode is not supported, the port driver automatically uses asynchronous mode.

To use synchronous mode in a transfer, the programmer of a SCSI class driver must ensure that both the SCSI port and the SCSI device involved in the transfer support synchronous mode. The SCSI port of the MicroVAX 3520/3540 systems supports both synchronous and asynchronous transfers, whereas that of the MicroVAX/VAXstation 3100 supports only asynchronous transfers.

To set up a connection to use synchronous data transfer mode, the SCSI class driver specifies a nonzero value in the **synchronous** longword of the connection characteristics buffer, and invokes the SPI$SET_CONNECTION_CHAR macro. The driver can also control the protocol of synchronous data transfers by supplying nonzero values for the **transfer period** and **REQ-ACK offset** longwords of the connection characteristics buffer and invoking the macro.

17.5.2 Enabling Disconnection and Reselection

The ANSI SCSI specification defines a disconnection facility that allows a target device to yield ownership of the SCSI bus while seeking or performing other time-consuming operations. When a target disconnects from the SCSI bus, it sends a sequence of messages to the initiator that cause it to save the state of the I/O transfer in progress. Once this is done, the target releases the SCSI bus. When the target is ready to complete the operation, it reselects the initiator and sends to it another sequence of messages. This sequence uniquely identifies the target and allows the initiator to restore the context of the suspended I/O operation.

Whether disconnection should be enabled or disabled on a given connection depends on the nature and capabilities of the device involved in the transfer, as well as on the configuration of the system. In configurations where there is a slow device present on the SCSI bus, enabling disconnection on connections that transfer data to the device can increase bus throughput. By contrast, systems where most of the I/O is directed toward a single device for long intervals can benefit from disabling disconnection. By disabling disconnection when there is no contention on the SCSI bus, port drivers can increase throughput and decrease the processor overhead for each I/O transfer.

By default, the VMS class/port interface disables the disconnect facility on a connection. To enable disconnection, the SCSI class driver sets bit 0 of the **connection flags** longword of the connection characteristics buffer, and invokes the SPI$SET_CONNECTION_CHAR macro.

17.5.3 Determining the Maximum Data Transfer Size

There are two factors governing the maximum data transfer size that any given SCSI device can accommodate.

First, there is the maximum size supported by the device; this can be determined from an inspection of the device's functional specification. The SCSI class driver writes the maximum device byte count to the device's UCB (UCB$L_MAXBCNT), usually by invoking the DPT_STORE macro when initializing the driver prologue table (DPT).

Secondly, there is the maximum value supported by the SCSI port. The port driver returns this value to the class driver in response to the class driver's invocation of the SPI$CONNECT macro.

The class driver may need to adjust the value in UCB$L_MAXBCNT to reflect the smaller of the device-specific and port-specific values.

The class driver compares the value supplied in IRP$L_BCNT with UCB$L_MAXBCNT to determine whether to accept, reject, or segment an I/O data transfer request.

17.5.4 Initializing the SCDRP to Reflect Class Driver Data Buffering Mechanisms

A standard data transfer, using direct I/O, involves the buffer specified in the $QIO system service call as the source or destination of the data involved in the transfer. Typically this buffer is in process space (P0 space) and mapped by the process's P0 page table. To access this buffer at elevated IPL, a driver calls a VMS-supplied FDT routine (such as EXE$READ or EXE$MODIFY) that locks the buffer into memory and returns the system virtual address of the first P0 page-table entry that maps the buffer. The servicing of the QIO_INQUIRY SCSI command by the SCSI class driver template follows this approach. (Note that the QIO_INQUIRY command is the means by which the template driver illustrates the transfer of data from a SCSI device to a process buffer. Ordinarily, for a specific SCSI device, a class driver would use a SCSI READ command.)

Other transfer operations may require that the class driver itself operate upon the contents of the data buffer, or maintain its own data buffer. For these operations, the class driver must allocate a system buffer from nonpaged pool. The servicing of the INQUIRY SCSI command by the SCSI class driver template follows this approach.

Depending upon the local buffering mechanism it uses to service an I/O request, a SCSI class driver must initialize the SCDRP with the parameters of the transfer. When a process buffer is involved in the transfer, the class driver initializes the following fields:

Field	Contents
SCDRP$L_ABCNT	0
SCDRP$W_FUNC	IRP$W_FUNC
SCDRP$W_STS	IRP$W_STS
SCDRP$L_MEDIA	IRP$L_MEDIA
SCDRP$L_SVAPTE	IRP$L_SVAPTE
SCDRP$W_BOFF	IRP$W_BOFF
SCDRP$L_BCNT	IRP$L_BCNT
SCDRP$L_PAD_COUNT	0
SCDRP$L_SCSI_FLAGS	SCDRP$V_SOBUF bit cleared

When a system buffer is involved in the transfer, the class driver initializes the following fields:

Field	Contents
SCDRP$L_SVA_USER	System virtual address of system buffer
SCDRP$L_SVAPTE	System virtual address of the system page table entry mapping the first page of the system buffer
SCDRP$L_BCNT	Length of the transfer
SCDRP$L_PAD_COUNT	0

Field	Contents
SCDRP$W_BOFF	Byte offset within page
SCDRP$W_STS	IRP$V_FUNC set for a read operation; clear for a write operation
SCDRP$L_SCSI_FLAGS	SCDRP$V_SOBUF bit set

17.5.5 Making a Class Driver Data Buffer Accessible to the Port

Regardless of the local buffering mechanism it requires to fulfill the I/O transfer, a SCSI class driver must make the buffer available to the SCSI port hardware. A SCSI class driver accomplishes this by invoking the SPI$MAP_BUFFER macro.

The SPI$MAP_BUFFER macro causes the SCSI port driver to reserve sufficient pages of the port's DMA buffer to accomplish the transfer, plus sufficient mapping resources, if required, to map the class driver's data buffer to system virtual addresses. Certain ports require this mapping so that the port driver can access a process space buffer when setting up or completing a transfer for the SCSI port. When the class driver initiates a write operation, the SCSI port driver uses its mapping resources to copy the data from the class driver's user or system data buffer to this intermediate DMA buffer, from which the SCSI port can access it. When the class driver initiates a read operation, the SCSI device transfers the data to the DMA buffer, from which the port driver copies it to the class driver's data buffer.

Other ports do not require this mapping and can access the class driver's data buffer using system page-table entries.

By convention, a SCSI class driver sets the SCDRP$V_BUFFER_MAPPED bit in SCDRP$L_SCSI_FLAGS when it invokes SPI$MAP_BUFFER to map a buffer; if the buffer involved in the transfer is a system buffer, it also sets the SCDRP$V_SOBUF bit. The SCDRP$V_SOBUF flag prevents the SPI$MAP_BUFFER port routine from double-mapping a system buffer.

The SPI$MAP_BUFFER port routine initializes the following fields in the SCDRP:

Field	Contents
SCDRP$L_SVA_USER	System virtual address of the system buffer. When the class driver's local buffer is a system buffer, the contents of this field are unchanged by SPI$MAP_BUFFER.
SCDRP$L_SPTE_SVAPTE	System virtual address of the system page-table entry mapping the first page of the system buffer. When the class driver's local buffer is a system buffer, the contents of this field and SCDRP$L_SVAPTE are identical.

Field	Contents
SCDRP$W_NUMREG	Number of pages of the port's DMA buffer allocated for this transfer.
SCDRP$W_MAPREG	Starting page number of the first DMA buffer page allocated for this transfer.

Once the SCSI command has been prepared, the SCSI class driver issues the command to the SCSI device by invoking the SPI$SEND_COMMAND macro.

When the data transfer has completed (or its failure has been serviced) and the port DMA buffer and mapping resources are no longer required, the class driver invokes the SPI$UNMAP_BUFFER macro to deallocate these resources.

17.5.6 Examining Port and SCSI Status

Whether a SCSI command completes or fails, the port driver returns to the class driver the following status values:

- Port status in R0
- SCSI command status in the low byte of the status buffer pointed to by SCDRP$L_STS_PTR
- Actual number of bytes transferred in SCDRP$L_TRANS_CNT

The class driver should examine these returned values to determine the success or failure of a SCSI command. If a SCSI command fails, the class driver can pursue its recovery or retry the command, depending upon the type and severity of the error and the nature of the device.

17.5.6.1 Examining Port Status

The port status is the primary indicator of the failure of a SCSI command; that is, if the port failed during command preparation or transmission, it is unlikely that the SCSI command status byte contains meaningful information.

The port driver returns one of the following status values in R0:

Status	Meaning
SS$_NORMAL	Normal successful completion
SS$_TIMEOUT	Failed during selection or arbitration
SS$_CTRLERR	Controller error or port hardware failure
SS$_BADPARAM	Bad parameter specified by the class driver
SS$_LINKABORT	Connection no longer exists
SS$_DEVACTIVE	Command outstanding on this connection

If R0 contains anything but success status, the class driver may want to examine it for specific status values and attempt error recovery, retry the operation, or return a special error status to the original $QIO call. At

the very least, the class driver should log a device error, according to the method described in Section 17.6.2.

17.5.6.2 Examining the SCSI Status Byte

If the port driver returns SS$_NORMAL status in R0, the class driver should proceed to check the SCSI command status in the low byte of the longword buffer pointed to by SCDRP$L_STS_PTR.

The format of a SCSI status byte is illustrated in Table 17–3. Interpretation of the bits in this status byte is device-specific. The VMS SCSI template driver (SKDRIVER) first clears reserved bits 0, 6, and 7. It compares the resulting value with the CHECK CONDITION status value, to determine if CHECK CONDITION status has been returned.

Table 17–3 SCSI Status Byte Format

Bits of Status Byte[1]								Status Represented
7	6	5	4	3	2	1	0	
R	R	0	0	0	0	0	R	GOOD
R	R	0	0	0	0	1	R	CHECK CONDITION
R	R	0	0	0	1	0	R	CONDITION MET/GOOD
R	R	0	0	1	0	0	R	BUSY
R	R	0	1	0	0	0	R	INTERMEDIATE/GOOD
R	R	0	1	0	1	0	R	INTERMEDIATE/CONDITION MET/GOOD
R	R	0	1	1	0	0	R	RESERVATION CONFLICT
R	R	1	0	1	0	0	R	QUEUE FULL (not implemented)

[1] All other codes reserved.

When CHECK CONDITION status is returned, SKDRIVER initiates a REQUEST SENSE SCSI command to determine the specific nature of the SCSI error. To do so, it must save the address of the SCDRP associated with the original command (in an extension to the device UCB), and allocate a new one for use with the REQUEST SENSE command. It prepares and issues the command according to the procedures described in Section 17.4. When the port driver returns status from the REQUEST SENSE command, SKDRIVER examines its status. If the port returns failure status or if the SCSI status byte has any error bit set, SKDRIVER completes the I/O request, deallocating both SCDRPs and its command and data buffers; and returns error status to the $QIO system service.

If the port returns success status from the REQUEST SENSE command, SKDRIVER examines the request sense key in its local system buffer (at SCDRP$L_SVA_USER). The actions of any class driver in response to any specific request sense key are device specific. SKDRIVER merely translates the value into a VMS success or failure status code and returns this code in R0. For sense keys indicating fatal errors, SKDRIVER logs a device error.

17.5.6.3 Testing for Incomplete Transfers

If both the port status value and the SCSI command status byte indicate successful completion, the class driver performs one last test to determine the success of any data transfer associated with the SCSI command.

The port driver returns the actual number of bytes transferred during command processing in SCDRP$L_TRANS_CNT. The class driver should compare the value in this field with the requested transfer size in SCDRP$L_BCNT. If they are not equal, the class driver may return successfully or investigate a possible error.

17.6 Other SCSI Class Driver Issues

The writer of a third-party SCSI class driver must deal with several issues that are not specifically related to the tasks of setting up a SCSI command or data transfer, but rather relate to the definition of the class/port interface. Among these issues are the following:

- Preserving the local context of the driver across calls to the port driver
- Logging errors detected by the class driver

Subsequent sections discuss each of these issues in detail.

17.6.1 Preserving Local Context

VMS SCSI port drivers contain routines that execute in response to a class driver's invocation of an SPI macro. A class driver should take into account the fact that any SPI macro invocation may cause the port driver routine to fork or stall while waiting for a port resource, and return to its caller's caller. These actions eradicate the local context of the class driver at the time it invoked the macro.

Therefore, a SCSI class driver routine must take special steps to ensure the following:

- The address of its caller is on the top of the stack.
- All significant local context currently in registers is preserved.
- Any local return address currently on the stack is preserved.

The SCSI class driver template (SKDRIVER) resolves these needs by allocating a 10-longword stack within its extension to the SCSI device UCB. The symbol UCB$L_STACK_PTR functions as a stack pointer. The class driver template defines macros that initialize the UCB stack (INIT_UCB_STACK), push and pop registers (or data) from the UCB stack (SUBPUSH and SUBPOP), push the return address from the top of the interrupt stack onto the UCB stack (SUBSAVE), and pop the return address from the UCB stack onto the interrupt stack and RSB (SUBRETURN).

Caution: The class driver must be careful not to overflow its local stack. Unless it takes precautions, it could overwrite data integral to a transfer in progress and cause unpredictable results.

Prior to calling any routine that may destroy its context, the class driver template issues a SUBSAVE to preserve its return address (before any additional data is pushed on the interrupt stack), and invokes the SUBPUSH macro for each register that must be preserved across the call. When execution in the class driver resumes, the driver issues the SUBPOP macro to restore the saved registers and the SUBRETURN macro to return to its caller.

17.6.2 Error Logging

A SCSI class driver establishes error logging and uses the system error logging routines (ERL$DEVICERR, ERL$DEVICTMO, and ERL$DEVICEATTN) as described in the *VMS Device Support Reference Manual.*

The VMS SCSI class/port interface defines SCSI port-driver and SCSI class-driver extensions to the error message buffer, which are interpreted and formatted by the VMS Error Log Utility (see Section 17.11.2). A SCSI class driver and the associated port driver log errors independently, each supplying SCSI-specific information as defined in its extension to the error message buffer.

The class driver extension to the error message buffer includes the information listed in Table 17–4.

Table 17–4 Error Message Buffer Extension for SCSI Class Drivers

Field	Length (in bytes)	Contents
Longword count	4	Number of longwords that follow in the error message buffer (not including this one).
Revision	1	Revision level of the error message buffer. The class driver must set this field to 1.
Hardware revision	4	Hardware revision information, returned by the SCSI INQUIRY command in ASCII format.
Error type	1	Type of error detected by the class driver. A SCSI class driver defines device-specific error types according to the nature of the device it services. The following error types are used by the VMS disk and tape class drivers and, as such, have defined values that are interpreted by the VMS Error Log Utility:

Error	Name	Description
01	CON_ERR	Attempt to connect to the port driver failed.
02	MAP_ERR	Attempt to map a user buffer failed.
03	SND_ERR	Attempt to send a SCSI command failed.
04	INV_INQ	Invalid inquiry data was received.

(continued on next page)

Table 17–4 (Cont.) Error Message Buffer Extension for SCSI Class Drivers

Field	Length (in bytes)	Contents		
		Error	**Name**	**Description**
		05	EXT_SNS_DAT	Extended sense data was returned from the SCSI device.
		06	INV_MOD_SNS	Invalid mode sense data returned from the SCSI device.
		07	REASSIGN_BLK	Reassign block.
		08	DIAG_DATA	Invalid diagnostic data returned to the VMS SCSI tape class driver.
SCSI ID	1	SCSI ID of the device to which the current command was sent. The SCSI ID is an integer between 0 and 7.		
SCSI LUN	1	SCSI LUN of the device to which the current command was sent. The SCSI LUN is an integer between 0 and 7.		
SCSI SUBLUN	1	Not used. This field always contains 0.		
Port status	4	Longword status returned in R0 from the port driver. A value of –1 in this field indicates that there is no valid data in this field.		
SCSI CMD	*n*	Current SCSI command bytes. The SCSI command bytes are preceded by a byte containing the length of the command.		
SCSI STS	1	Current SCSI status byte. A status byte of –1 in this field indicates that the status byte does not yet contain valid information.		
Additional data	*n*	Additional data, preceded by a byte count of the data. A class driver defines what additional data would be meaningful in an error log entry based on the type of device it services. Additional data is displayed by the VMS Error Log Utility as untranslated longwords.		

Prior to calling ERL$DEVICERR to log an error associated with device activity, or ERL$DEVICEATTN (or ERL$DEVICTMO) to log an error on an inactive device, the class driver should perform the following tasks:

- Ensure that the DDT$W_ERRORBUF field contains a sufficient byte count to accommodate both the standard error message buffer and the SCSI class driver extension. The class driver can either supply this value in the **erlgbf** argument to the DDTAB macro or specifically initialize DDT$W_ERRORBUF.
- Initialize the device type (UCB$B_DEVTYPE) and device class (UCB$B_DEVCLASS) fields to DT$_GENERIC_SCSI and DC$_MISC. A driver normally initializes these fields by invoking the DPT_STORE macro. If the driver must use other device type or class values, or allows them to be changed by a user program, it may need to save and restore the real values of these fields temporarily across calls to the error logging routines.

ERL$DEVICERR, ERL$DEVICTMO, and ERL$DEVICEATTN all result ultimately in a call to the class driver's register dumping routine. The register dumping routine must supply all available information about the SCSI device error in the class driver's SCSI-specific extension to the error message buffer.

The VMS SCSI class driver template (SKDRIVER) defines the fields of this extension and contains a macro (LOG_ERROR) and a routine (ERROR_LOG) that are a useful basis for the implementation of error logging in a third-party SCSI class driver.

17.7 Flow of a Read I/O Request Through the SCSI Class and Port Drivers

This section describes a hypothetical read-I/O request to a SCSI device as it is serviced by a SCSI class driver and the port driver. The discussion assumes that the read operation is successful.

When it is loaded, the class driver performs a one-time initialization sequence as follows:

1. Its unit initialization routine invokes the SPI$CONNECT macro. In response, the port driver forms a logical connection between the SCSI device's UCB and the target on the SCSI bus. The port driver creates an SCDT in which it inserts information describing the connection.
2. Its unit initialization routine optionally invokes the SPI$SET_CONNECTION_CHAR macro to set the appropriate data transfer mode or timeout values, or to enable disconnection of the connection. In response, the port driver modifies the connection-specific characteristics it maintains in the SCDT.

When the class driver receives a read-I/O request, it performs the following operations:

1. Its read FDT routine verifies and interprets the parameters of the $QIO system service call.
2. It calls a system FDT routine that locks the specified process buffer in memory.
3. When the request becomes current, the start-I/O routine dispatches to code that services the specified function.
4. Its start-I/O routine allocates and initializes an SCDRP and copies to it the fields from the IRP required by the port driver to complete the read operation.
5. Its start-I/O routine invokes the SPI$ALLOCATE_COMMAND_BUFFER macro. In response, the port driver allocates a buffer suitable for a SCSI command descriptor buffer and a SCSI status byte.
6. Its start-I/O routine invokes the SPI$MAP_BUFFER macro. In response, the port driver allocates the resources required to make the process buffer available to the port driver.
7. Its start-I/O routine builds a SCSI command in the command descriptor buffer, initializes the status byte, and invokes the SPI$SEND_COMMAND macro to send the command to the port driver.

When the port driver receives the command, it sets up the connection characteristics (data transfer mode, timeout value, and disconnect mode) recorded in the SCDT, sends the command buffer to the device, and responds to changes in SCSI bus phases. The port driver performs the following specific actions:

1 It requests and obtains ownership of the port, stalling if necessary until the port is available.

2 It arbitrates for ownership of the SCSI bus.

3 It selects a target device on the SCSI bus and sends it an IDENTIFY message.

4 It waits for the bus COMMAND phase.

5 It sends the SCSI command descriptor buffer, byte by byte, to the target device.

6 It waits for a SCSI bus phase change. If the next phase is not DATA IN, the port driver proceeds with the next step. Otherwise, it accepts data from the target device as follows:

   a. It sets up and starts a DMA transfer to the port's DMA buffer.

   b. It saves its context in the port UCB and waits for the target device to interrupt, signifying the completion of the read request. If the target device does not interrupt, the port driver sets up error status and returns to the class driver.

7 It checks the SCSI bus phase. If the phase is unchanged, the port driver sets up the next transfer. If the phase is STATUS, the port driver reads the status and copies the status to the return status buffer.

8 It waits for the MESSAGE IN phase. When the phase changes to MESSAGE IN, the port driver reads the message. If the message is COMMAND COMPLETE, the port driver returns SS$_NORMAL in R0. Otherwise, it returns the appropriate port status to the class driver.

9 It releases the port.

10 It transfers the data from the port's DMA buffer to the process buffer.

11 It returns to the class driver.

When it regains control from the port driver, the class driver performs the following tasks to complete the read operation:

1 It checks the port status in R0.

2 It checks the SCSI status in the SCSI status byte.

3 It checks that the actual transfer length agrees with the requested transfer length.

4 It invokes the SPI$DEALLOCATE_COMMAND_BUFFER macro to deallocate the command buffer.

5 It invokes the SPI$UNMAP_BUFFER macro to release the port resources mapping the user buffer.

6 It initiates device-independent postprocessing of the request by invoking the REQCOM macro.

17.8 Components of a SCSI Class Driver

A SCSI class driver contains nearly all of the components of a traditional VMS driver. These include the following:

- Data, macro, and constant definitions
- Driver prologue table
- Driver dispatch table
- Function decision table and FDT routines
- Controller initialization routine
- Unit initialization routine
- Start-I/O routine
- Cancel-I/O routine
- Error logging routine
- Register dumping routine

A SCSI class driver contains no interrupt service routine. Moreover, it has no access to device control and status registers (CSRs). It relies on the port driver to initiate operations on the device and to service device interrupts.

This section describes the special operations that must be performed by the components of a SCSI class driver. The standard and typical operations performed by driver routines and tables are discussed in Part II.

17.8.1 Data Definitions

A SCSI class driver must invoke the $SCDRPDEF data structure definition macros, located in SYS$LIBRARY:LIB.MLB. $SCDRPDEF defines the fields of the SCSI class driver request packet.

A SCSI class driver typically does not reference fields in the SCSI connection descriptor table and, thus, does not need to invoke the $SCDTDEF macro. Although fields in the SCSI port descriptor table are used by the SPI macros as vectors to routines in the port driver, a SCSI class driver need not explicitly define SPDT fields. It indirectly obtains the SPDT definitions through its invocation of the SPI$CONNECT macro; it is the macro that invokes $SPDTDEF.

A SCSI class driver may define an extension to the device UCB for an internal stack or for managing the allocation of SCDRPs, depending upon the needs of the implementation. The VMS SCSI template driver (SKDRIVER), listed in Appendix B, illustrates uses of these additional

UCB fields. SKDRIVER also defines symbols representing SCSI-specific data buffer offsets and status values.

17.8.2 Driver Prologue Table

A SCSI class driver must supply the NULL keyword as the **adapter** argument to the DPTAB macro. It also must specify that the DPT$V_NO_IDB_DISPATCH flag is set in the **flags** argument. The DPT$V_NO_IDB_DISPATCH flag indicates that the IDB$L_UCBLIST field is not used to store the addresses of UCBs for this device.

If the class driver implements error logging, it should use the DPT_STORE macro to initialize UCB$B_DEVTYPE to DT$_GENERIC_SCSI and UCB$B_DEVCLASS to DC$_MISC. If the class driver must use other device type or class values, or allows them to be changed by a user program, it may need to save and restore the real values of these fields temporarily across calls to the error logging routines.

A SCSI class driver should not initialize CRB$L_INTD+VEC$L_ISR or the other interrupt vectors in the CRB. A SCSI device interrupts through a vector serviced by the port driver; any interrupt service routine specified by the SCSI class driver is not used.

17.8.3 Driver Dispatch Table

There are no special requirements for a SCSI class driver's driver dispatch table.

17.8.4 Function Decision Table and FDT Routines

There are no special requirements for a SCSI class driver's function decision table.

A class driver invokes FDT routines to preprocess I/O functions in a device-specific manner. Most SCSI class drivers use the standard VMS-supplied FDT routines (such as EXE$READ, EXE$WRITE, and EXE$SETMODE). However, some class drivers may need to include a special FDT routine. The VMS SCSI class driver template illustrates this approach.

17.8.5 Controller Initialization Routine

There are no special requirements for a SCSI class driver's controller initialization routine.

17.8.6 Unit Initialization Routine

A SCSI class driver's unit initialization routine must perform several special actions, as follows:

- It checks the power failure bit (UCB$V_POWER) in UCB$W_STS to determine whether it is being called in the course of power failure recovery. If this bit is set, the unit initialization routine returns immediately.
- It forks twice, issuing the FORK macro twice in succession. The first fork ensures, during system initialization or autoconfiguration, that the SCSI port driver's initialization routines begin execution before the class driver performs its initialization. The second fork guarantees that a port driver initialization fork thread has created its SPDTs and initialized the SCSI ports.

  Note that the unit initialization routine must be executing at fork IPL when it invokes the SPI$CONNECT macro.
- It prepares for an SPI$CONNECT request by obtaining the SCSI port ID, the SCSI device ID, and the SCSI logical unit number (LUN).

  SCSI device unit numbers have the form *d0u*, where *d* is the device ID and *u* is the LUN. The unit initialization routine obtains the SCSI device unit number from UCB$W_UNIT and divides it by 100 (using the EDIV instruction). The quotient (in R1) is the device ID and the remainder (in R2) is the LUN. Both should be values between 0 and 7.

  SCSI port IDs are represented by the alphabetic characters *A* and *B*. The unit initialization routine obtains this letter from the third byte in DDB$T_NAME (for instance, *A*, from SKA500) and converts it to the numeral 0 or 1.

  Once it has obtained the SCSI port ID, the SCSI device ID, and the SCSI LUN, the unit initialization routine sets up the registers for the call to SPI$CONNECT as follows:

ID	Destination
SCSI port ID	Low-order word of R1
SCSI device ID	High-order word of R1
SCSI LUN	High-order word of R2

- It invokes the SPI$CONNECT macro. The port driver, as a result, attempts to create a connection between the class driver and the port.

  If the class driver expects notification of asynchronous events from the target device, it supplies the address of a local selection callback routine in the **callback** argument of the SPI$CONNECT macro. (For a discussion of asynchronous event notification (AEN) mode, see Section 17.9.)

  — If the port driver returns failure status, the unit initialization routine sets the device off line.

— If the port driver successfully creates the connection, the unit initialization routine initializes UCB$L_MAXBCNT, UCB$L_SCDT, and UCB$L_PDT with the values returned by the port driver, sets the device unit on line (by setting UCB$V_ONLINE in UCB$W_STS), and returns success status to its caller.

The VMS template SCSI class driver (SKDRIVER) unit initialization routine performs such optional actions as setting up an internal stack in the UCB for context-saving purposes, and allocating nonpaged pool for a set of SCDRPs to be queued to the UCB for use by the driver's start-I/O routine. See the Appendix B for a listing of the SCSI class driver template.

The unit initialization routine may also invoke the SPI$GET_CONNECTION_CHAR and SPI$SET_CONNECTION_CHAR macros to examine (and possibly alter) the current data transfer mode, timeout, command retry, and disconnect characteristics of the SCSI connection. (See Section 17.5 for additional information.)

17.8.7 Start-I/O Routine

A SCSI class driver's start-I/O routine must perform the following steps to prepare a SCSI command for delivery to the port driver:

- Allocate an SCDRP from nonpaged pool. (The VMS template SCSI class driver allocates SCDRPs in its unit initialization routine; its start-I/O routine simply removes a preallocated SCDRP from a queue in the device UCB.)
- Insert the address of the IRP in SCDRP$L_IRP.
- Dispatch to a function-specific command preparation routine.

The command preparation routine performs the procedures described in Section 17.4 and Section 17.5. Its actions typically involve the following:

- Invoking SPI$ALLOCATE_COMMAND_BUFFER to allocate a port command buffer in which it assembles the SCSI command and reserves a longword for the SCSI status byte to be returned from command execution.
- Initializing fields in the SCDRP from the corresponding fields in the IRP. For read-I/O functions, the class driver must ensure that IRP$V_FUNC is set in SCDRP$W_STS.
- Invoking SPI$MAP_BUFFER to make data in the process buffer available to the port, and setting the SCDRP$V_BUFFER_MAPPED bit in SCDRP$L_SCSI_FLAGS to indicate that the buffer has been mapped to the port. (If it maps a system buffer, it must set both the SCDRP$V_S0BUF and the SCDRP$V_BUFFER_MAPPED bits.)
- Invoking SPI$SEND_COMMAND to deliver the SCSI command to the port driver.
- When the command completes, examining the port status, SCSI status, and transfer count to determine the success or failure of the I/O operation. (See Section 17.5.6 for a detailed description of the means by which a SCSI class driver typically responds to status information.)

If the operation fails, the class driver may take steps to obtain additional status information from the target device, pursue error recovery and retry the operation, enter a device-specific message in the error log buffer, return error status to the $QIO system service, or perform some combination of these actions.

- Invoking SPI$UNMAP_BUFFER to release port mapping resources.
- Invoking SPI$DEALLOCATE_COMMAND_BUFFER to deallocate the port command buffer.
- Deallocating the SCDRP.
- Initiating device-independent postprocessing by invoking the REQCOM macro.

17.8.8 Cancel-I/O Routine

If a SCSI class driver receives a cancel request for an I/O operation in progress on a SCSI device, its cancel-I/O routine may invoke the SPI$ABORT_COMMAND macro to terminate the I/O operation.

Note: VAXstation 3520/3540 systems do not implement the abort-SCSI-command function.

17.8.9 Register Dumping Routine

A SCSI class driver's register dumping routine executes in the course of a driver error logging operation. The class driver calls ERL$DEVICERR, ERL$DEVICTMO, or ERL$DEVICEATTN, and the system error logging routine calls the driver's register dumping routine.

The register dumping routine loads the error message buffer with all available information about a SCSI device error in the buffer extension reserved for SCSI class driver information (see Table 17–4). Detailed information on SCSI class driver error logging appears in Section 17.6.2.

17.9 Servicing Asynchronous Events from a SCSI Device

Devices can perform one of two roles on the SCSI bus; either the target role or the initiator role. Typically, the host processor serves as the initiator and peripheral devices serve as targets. However, some devices require that the host processor respond to an unsolicited event, such as when the device is selected or deselected. When such an event occurs, the target device must be capable of selecting the host and acting in the initiator mode.

Certain MicroVAX/VAXstation systems implement the SCSI asynchronous event notification (AEN) feature, allowing SCSI devices to act as initiators on given connections. When AEN is enabled and the host is selected by a target, the host

- Responds to selection
- Parses SCSI command packets

- Drives the SCSI bus phase as required by targets

The VMS SCSI class/port interface supports asynchronous event notification by the SPI$CONNECT macro and an extension to the SCSI port interface (SPI). Table 17–5 lists the SPI macros provided in the SPI extension.

Table 17–5 SPI Extension Macros Supporting Asynchronous Event Notification

SPI$FINISH_COMMAND[1]	Completes an I/O operation executing under the AEN feature
SPI$RECEIVE_BYTES	Receives command, message, and data bytes from a device acting as an initiator
SPI$RELEASE_BUS[1]	Releases the SCSI bus
SPI$SEND_BYTES	Sends command, message, and data bytes to a device acting as an initiator
SPI$SENSE_PHASE	Reads the current SCSI bus phase
SPI$SET_PHASE	Sets the SCSI bus phase

[1]A SCSI class driver must invoke either the SPI$FINISH_COMMAND macro or the SPI$RELEASE_BUS macro (but not both) to complete an AEN operation.

To utilize asynchronous event notification, a SCSI class driver's unit initialization routine must provide the address of a selection callback routine in the call to the SPI$CONNECT macro. The port driver invokes the callback routine at this address in response to selection by another device. The port driver passes the selection callback routine the address of the SPDT in R4 and any optional selection context in R5. When invoked, the callback routine is executing at IPL 8, holding the fork lock.

If the SCSI class driver does not provide a callback address, no selections are allowed on the connection that is established. If a selection does occur on a connection that is not set up to accommodate selections, the port driver releases the SCSI bus.

The flow of an AEN operation is as follows:

1 The class driver connects to the port driver and provides the callback address.

2 The port driver receives a selection on an existing connection. If selections are allowed, the port driver calls the class driver at its callback address, holding the fork lock at IPL 8. R4 contains the address of the SCDT and R5 contains the address of the SCDRP.

3 The class driver invokes SPI$SET_PHASE to set the SCSI bus to COMMAND phase.

Because the target has selected the host, the host now becomes the target. In SCSI, the target drives the phase of the SCSI bus after selection. Thus, the class driver drives the SCSI bus to the COMMAND phase to receive the command bytes from the initiator.

4 The class driver invokes SPI$RECEIVE_BYTES.

Because command packets are variable in size, the class driver requests the first byte of the command to determine how many additional command bytes are to be expected.

5 The class driver again invokes SPI$RECEIVE_BYTES.

Once the class driver has determined exactly how many command bytes are expected, it requests all remaining command bytes with this one call.

6 The class driver invokes SPI$SET_PHASE to set the SCSI bus phase to DATA IN.

After the class driver has received all the command bytes and parsed the command to identify it, the class driver sets the bus to the appropriate phase. For instance, if the command is READ BUFFER, the class driver must set the bus phase to DATA IN.

7 The class driver invokes SPI$SEND_BYTES to return exactly the number of data bytes requested by the initiator.

8 The class driver invokes SPI$FINISH_COMMAND.

9 The class driver returns from its callback routine.

Once the data transfer has completed, the I/O operation is done from the class driver's perspective. The class driver completes the I/O operation by returning status and the COMMAND COMPLETE message to the device.

17.10 Configuring a SCSI Third-Party Device

The System Generation Utility (SYSGEN) loads a third-party SCSI class driver into system virtual memory, creates additional data structures for the device unit, and calls the driver's controller initialization routine and unit initialization routine. SYSGEN automatically loads and autoconfigures the SCSI port driver at system initialization. As part of autoconfiguration, SYSGEN polls each device on each SCSI bus. If the device identifies itself as a direct-access device, a direct-access CDROM device, or a flexible disk device, SYSGEN automatically loads the VMS disk class driver (DKDRIVER); if the device identifies itself as a sequential-access device, SYSGEN automatically loads the VMS tape class driver (MKDRIVER). If the autoconfiguration facility does not recognize the type of the SCSI device, it loads no driver.

Consequently, third-party SCSI devices must be configured and their drivers loaded by an explicit SYSGEN CONNECT command, as follows:

```
$ RUN SYS$SYSTEM:SYSGEN
SYSGEN> CONNECT mmpd0u /NOADAPTER
```

In this command, *mm* represents the device mnemonic (for instance, SK; *p* represents the SCSI port ID (for instance, the controller ID *A* or *B*); *d* represents the SCSI device ID (a digit from 0 to 7); 0 signifies the digit zero; and *u* represents the SCSI logical unit number (a digit from 0 to 7).

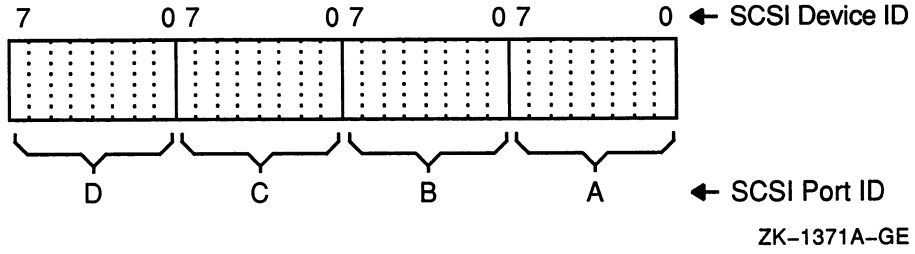
17.10.1 Disabling the Autoconfiguration of a SCSI Device

Note that, in special cases, you may need to prevent SYSGEN's autoconfiguration facility from loading the VMS disk or tape class driver for a device with a specific port ID and device ID. This would be the case if a third-party device should identify itself as either a random-access or sequential-access device and were to be supported by the VMS generic SCSI class driver.

To disable the loading of a VMS disk or tape driver for any given device ID, VMS temporarily defines the special SYSGEN parameter **SCSI_NOAUTO**.

The **SCSI_NOAUTO** system parameter, as shown in Figure 17–5, stores a bit mask of 32 bits in which the low-order byte corresponds to the first SCSI bus (PKA0), the second byte corresponds to the second SCSI bus (PKB0), and so on. For each SCSI bus, setting the low-order bit inhibits automatic configuration of the device with SCSI device ID 0; setting the second low-order bit inhibits automatic configuration of the device with SCSI device ID 1, and so forth. For instance, the value 00002000_{16} would prevent the device with SCSI ID 5 on the bus identified by SCSI port ID *B* from being configured. By default, all of the bits in the mask are cleared, allowing all devices to be configured.

Figure 17–5 SCSI_NOAUTO System Parameter



Note: The VMSD2 system parameter is now a special parameter reserved to Digital.

17.11 Debugging a SCSI Class Driver

VMS device drivers execute in kernel mode at elevated interrupt priority levels. Problems in device driver code often manifest themselves in system failures and system hangs. Chapter 13 describes some general methods for debugging device drivers that can also be used to debug a third-party SCSI class driver. While using the XDELTA debugger to investigate problems in a class driver, however, you should set breakpoints such that you can easily step over VMS SCSI port driver code.

As discussed in Section 17.6.2, Digital strongly recommends that a third-party SCSI class driver respond to port and SCSI status return values, and that it incorporate an error logging routine that records events significant to the device. Class driver error log entries, as well as VMS port driver error log entries, can provide clues that are helpful in resolving problems (see Section 17.11.2) that may occur during the development of a third-party SCSI class driver.

Among the problems that commonly occur in early versions of SCSI class drivers are the following:

- The class driver has failed to deallocate a port resource, such as a command buffer or port map registers. You should ensure that the class driver invokes the SPI$DEALLOCATE_COMMAND_BUFFER and the SPI$UNMAP_BUFFER macros before completing a data transfer (that is, before invoking the REQCOM macro).
- The class driver has sent a SCSI command to a device, but the device does not support the command. Typically, the device times out or the port driver logs an entry for a bad phase transition event.
- The class driver has sent a misformatted SCSI command packet to a device. This problem also results in a device timeout or phase error.

Hardware problems on a SCSI bus can cause a SCSI command to fail, regardless of whether the device to which the command was directed is at fault. When testing and debugging a class driver for a new device on a SCSI bus, you should ensure that bus traffic from busy or faulty devices elsewhere on a SCSI bus does not interfere with the device's operation. Isolate the device by placing it on a SCSI bus reserved for it and the processor alone or, if that is not possible, by placing it on the SCSI bus on which the system disk does *not* reside.

17.11.1 Selecting a SCSI Bus Analyzer

Finally, in debugging a SCSI class driver, you may find a SCSI bus analyzer to be a valuable aid.

A SCSI bus analyzer is a passive device that monitors all traffic on the SCSI bus to which it is connected, and displays in a useful format the data it has collected. Some analyzers can be used in an active mode to generate packets on the bus; however, this is generally more useful to developers of SCSI target devices than to writers of class drivers.

A SCSI bus analyzer is commonly used to verify that the commands the class driver generates (or should generate) are actually being transmitted across the SCSI bus. The most useful analyzers can interpret the SCSI phase lines and display the phase along with the data sent during that phase. This helps the writer of a class driver pinpoint the location of a possible coding problem. Another common use of an analyzer is to capture infrequent errors such as bus hangs or a target dropping off the bus.

Some features to look for in an analyzer are as follows:

- Ability to interpret the bus phase lines and display the data according to the phase
- A "timing mode" that displays bus signals in the form of a timing diagram
- Ability to trigger the analyzer on a specific event, such as a specific data pattern in a specific phase or a bus reset
- Ability to dump the contents of the display to a printer

17.11.2 Interpreting SCSI Error Log Entries

As dictated by the VMS SCSI class/port driver architecture, a VMS SCSI port driver logs port-specific events in a defined form. Port driver error log entries can provide clues that are helpful in resolving problems that may occur during the development of a third-party SCSI class driver.

The VMS SCSI class/port driver architecture also specifies a form for class driver error log entries. Because of the value of the error log in debugging, Digital highly recommends that a third-party SCSI class driver incorporate an error logging routine that records events significant to the device. (See Section 17.6.2 for a discussion of the procedures by which class drivers interpret status, format events, and register error log entries.)

You can use the VMS Error Log Utility, as described in the *VMS Error Log Utility Manual*, to list and format SCSI port and class driver error log entries.

17.11.2.1 SCSI Port Driver Error Log Entries

The SCSI port driver is responsible for all low-level activity associated with sending commands to a target SCSI device. The standard format of an error log entry generated by a port driver has two parts: a port-common section and a port-specific section. All VMS port drivers provide the same type of information in the port-common section of the entry. The information a port driver supplies in the port-specific section depends upon the SCSI port hardware that is in use.

Table 17–6 describes the contents of a formatted port driver error log entry. A reference number in the table column "Field" associates each table item with an entry in the representative error logs presented in Examples 17–1 and 17–2.

When inspecting a SCSI port driver error log entry, first examine the **error type** and **error subtype**. These fields indicate the nature of the event that occurred. Also check the **SCSI ID** field to determine the device for which the event has been reported. Although the SCSI ID may not always identify the device responsible for the event, it may help you interpret the significance of the information in this and other error log entries.

Next, examine the **SCSI CMD** field to determine which SCSI command was current at the time of the logged event. The **phase queue** entry lists those SCSI bus phases that have been successfully completed during execution of this command. You can derive the current phase of the SCSI bus by referring to the description of the phase signals defined for the command in the ANSI SCSI specification. In addition, the port-specific section of the error log entry of certain VMS port drivers lists the currently asserted bus lines.

Finally, the sets of counters that appear in a port driver error log entry can help you discern patterns of activity on the SCSI bus. For instance, a large number of parity errors are a symptom of a bus termination problem or other hardware problem.

Table 17–6 Key to Port Driver Error Log Entries

Field[1]	Description
General Event Information	
Error type ❶ Error subtype ❷	Error type and subtype. The following types and subtypes are defined.

Error[2]	Definition	Description
01	BUS_HUNG	SCSI bus was continuously busy during an arbitration attempt.
02	ARB_FAIL	Arbitration of SCSI bus failed due to activity of higher priority devices.
03	SEL_FAIL	Selection failed.
04	TIMEOUT	Timeout occurred.
05	PARITY_ERROR	Parity error detected.
06	PHASE_ERROR	SCSI bus phase error. A phase error results from a missing SCSI bus phase, a phase that is entered more than once, or a bad phase sequence.

Subtype[2]	Description
01	Missing phase error
02	Bad phase transition
03	Timeout waiting for phase interrupt
04	Unexpected phase change during DATA IN; error during REQ-ACK
05	Unexpected phase change during DATA OUT; error during REQ-ACK
06	Phase change timeout during DATA IN
07	Phase change timeout during DATA OUT
08	Timeout waiting for phase change
09	Phase change timeout during COMMAND OUT
10	Bus freed during command phase

Error[2]	Definition	Description
07	BUS_RESET	Bus reset detected.

Subtype[2]	Description
01	Reset occurred while no I/O operation was active

Error[2]	Definition	Description
08	UNEXPECTED_ INTERRUPT	Unexpected interrupt received.
09	BUS_RESET_ ISSUED	Bus reset initiated.

[1]Reference numbers refer to Examples 17–1 and 17–2.

[2]Error type and subtype values are rendered in hexadecimal format.

(continued on next page)

Table 17–6 (Cont.) Key to Port Driver Error Log Entries

Field[1]	Description		
General Event Information			
	10	RESEL_ERR	Error following a device disconnect. **Subtype[2] Description** 01 Bad parity during reselect 02 No target ID during reselect 03 Multiple target IDs during reconnect 04 No connection to this target 05 Failed while no reselect was pending 08 SEL failed to clear during reselect 09 REQ failed to set during reselect 10 Bad RESEL message
	11	CTL_ERR	Error detected by controller.
	12	BUS_ERR	Controller detected a SCSI bus protocol error.
	13	ILLEGAL_MSG	Illegal message received.
	14–19		Reserved.
SCSI ID ❸	SCSI ID of the device to which the current command is being sent. Valid SCSI IDs range from 0 to 7. A value of FF_{16} in this entry indicates that the SCSI ID is unknown or not relevant (as in the case of a spurious bus reset).		
SCSI CMD ❹	Current SCSI command.		
SCSI MSG ❺	Current SCSI message.		
SCSI STATUS ❻	Current SCSI status. A status value of FF_{16} indicates that the SCSI bus has not yet returned status.		
Port Error Counters[3]			
Bus busy count ❼	Number of times the port driver has attempted to arbitrate for the SCSI bus and has found the bus hung for an extended period of time. A value in this field indicates either that the bus is extremely busy or a device on the bus is hung.		
Unsolicited reset count ❽	Number of times the port driver has received a reset interrupt that is not due to its own pulling of the bus reset line. This could be due to noise on the reset line or to a device (or another initiator) pulling the bus reset line.		
Unsolicited interrupt count ❾	Number of times the port driver has received an unsolicited interrupt.		

[1]Reference numbers refer to Examples 17–1 and 17–2.

[2]Error type and subtype values are rendered in hexadecimal format.

[3]The port error counters record errors that cannot be attributed to a specific device on the SCSI bus.

(continued on next page)

Table 17–6 (Cont.) Key to Port Driver Error Log Entries

Field[1]	Description
Connection Error Counters[4]	
Arbitration fail count ⑩	Number of times the port driver has attempted to arbitrate for the SCSI bus and has failed. Arbitration is attempted only when a bus free condition is detected. Thus, this counter reflects the number of times a low priority device loses arbitration to a higher priority device.
Selection fail count ⑪	Number of times the port driver has attempted to select a target device and has failed. This could happen just after a target device has been reset, if it has been powered off or disconnected from the bus, or if it is hung in such a way that it is not also hanging the bus.
Parity error count ⑫	Number of times the port driver has detected a parity error while sending a command to a target SCSI device.
Phase error count ⑬	Number of times the port driver has detected a phase error while sending a command to a target SCSI device.
Bus reset count ⑭	Number of times the port driver has reset the bus because it was unable to send a command to a target SCSI device. The port driver resets the bus for a number of reasons: for instance when it detects a bus hang or a phase error.
Bus error count ⑮	Number of times the SCSI controller has detected an error on the SCSI bus. This field is not used by SCSI controllers on MicroVAX/VAXstation 3100 systems.
Controller error count ⑯	Number of times the SCSI controller has reported an internal error. This field is not used by SCSI controllers on MicroVAX/VAXstation 3100 systems.
Retry Counters	
Arbitration retry count ⑰	Number of arbitration retries attempted. A value of –1 indicates that the counter contains no valid data.
Selection retry counter ⑱	Number of selection retries attempted. A value of –1 indicates that the counter contains no valid data.
Bus busy retry counter ⑲	Number of bus busy retries attempted. A value of –1 indicates that the counter contains no valid data.
Phase Queue	
Element phase queue ⑳	Lists the SCSI phases that have been entered and completed during the execution of the current command. The digit preceding the list indicates the number of phases that have been completed.
Port dependent data ㉑	Contents of port controller registers. This section of the error log entry contains information specific to the SCSI port controller employed by the system.

[1] Reference numbers refer to Examples 17–1 and 17–2.

[4] The connection error counters record errors that can be attributed to a specific device on the SCSI bus. The SCSI ID field specifies the devices to which the command was being sent when the error occurred.

17.11.2.2 SCSI Class Driver Error Log Entries

A SCSI class driver logs device-specific events in the manner described in Section 17.6.2. Although all class drivers use a common extension to the standard error message buffer when logging errors, the types of events detected and reported by class drivers are specific to the devices they control.

Table 17–7 describes the contents of a formatted class driver error log entry. Each item in the "Field" column of this table is associated with a field in the error log contained in Example 17–3.

Table 17–7 Key to Class Driver Error Log Entries

Field[1]	Description
Hardware revision ㉒	Hardware revision information, returned by the SCSI INQUIRY command.
Error type ㉓	Type of error detected by the class driver. A SCSI class driver defines device-specific error types according to the nature of the device it services. The following error values are interpreted by the VMS Error Log Utility:

Error[2]	Name	Description
01	CON_ERR	Attempt to connect to the port driver failed
02	MAP_ERR	Attempt to map a user buffer failed
03	SND_ERR	Attempt to send a SCSI command failed
04	INV_INQ	Invalid inquiry data was received
05	EXT_SNS_DAT	Extended sense data was returned from the SCSI device
06	INV_MOD_SNS	Invalid mode sense data returned from the SCSI device
07	REASSIGN_BLK	Reassign block
08	DIAG_DATA	Invalid diagnostic data returned to the VMS SCSI tape class driver

SCSI ID ㉔	SCSI ID of the device to which the current command was sent. The SCSI ID is an integer between 0 and 7.
SCSI LUN ㉕	SCSI logical unit number of the device to which the current command was sent. The SCSI LUN is an integer between 0 and 7.
SCSI SUBLUN ㉖	Not used. This field always contains 0.
Port status ㉗	Current port status. A value of –1 indicates that there is no valid data in this field.
SCSI CMD ㉘	Current SCSI command.
SCSI STS ㉙	Current SCSI status. A value of –1 indicates that there is no valid data in this field.

[1] Reference numbers refer to Example 17–3.

[2] Error type values are rendered in hexadecimal format.

(continued on next page)

Table 17–7 (Cont.) Key to Class Driver Error Log Entries

Field[1]	Description
Additional data ㉚	Additional data, preceded by a byte count of the data. A class driver defines what additional data would be meaningful in an error log entry based on the type of device it services. Additional data is displayed by the VMS Error Log Utility as untranslated longwords. Note that the VMS Error Log Utility can interpret extended sense data values when the extended sense data received error type is reported in the log and the driver's error logging routine places the sense data in this field of its error message buffer. Thus, the error log entry that appears in Example 17–3 interprets the logged sense data as a unit attention message signifying a power on or reset condition.

[1]Reference numbers refer to Example 17–3.

17.12 Resolving SCSI Class Driver Problems Using Error Logs

Taken as a unit, Examples 17–1 through 17–3 illustrate a standard event sequence that may occur during a SCSI bus transaction. This sequence involves the following actions:

1. The port detects an abnormal event, such as a timeout. (Example 17–1)
2. The port driver resets the SCSI bus. (Example 17–2)
3. A class driver receives extended sense data from the device informing it of the reset event. (Example 17–3)

These events typically occur for one of the following reasons:

- The class driver has sent a SCSI command to a device that the device does not understand or does not support.
- The class driver has sent a misformatted SCSI command packet to a device.
- The class driver has failed to deallocate a port resource, such as a command buffer or port map registers.
- A hardware failure has occurred on the SCSI bus.

Example 17–1 SCSI Bus Phase Error Port Driver Error Log Entry

```
V A X / V M S              SYSTEM ERROR REPORT     COMPILED 13-SEP-1990 15:05
                                                                 PAGE   1.

******************************* ENTRY     208. *******************************
ERROR SEQUENCE 43028.                        LOGGED ON:        SID 0A000005
DATE/TIME 13-SEP-1990 15:03:35.08                           SYS_TYPE 04010102
SCS NODE:                                                    VAX/VMS X5.2-1C

DEVICE ATTENTION  KA420  CPU REV# 6.

SCSI PORT SUB-SYSTEM, UNIT _PKA0:

      ERROR TYPE            06
                                        SCSI BUS PHASE ERROR(1)
                                        SUB-ERROR TYPE = 02(X)(2)
      SCSI ID               02
                                        SCSI ID = 2.(3)
      SCSI CMD         CA810208
                           0019
                                        READ(4)
      SCSI MSG              00
                                        COMMAND COMPLETE(5)
      SCSI STATUS           FF
                                        NO STATUS RECEIVED(6)
      PORT ERROR CNT   00000000
                       00000000
                       00000000
                                        BUS BUSY CNT = 0.(7)
                                        UNSOL RESET CNT = 0.(8)
                                        UNSOL INTRPT CNT = 0.(9)
      CONN ERROR CNT   00000000
                       00000000
                       00000000
                       00000001
                       00000000
                       00000000
                       00000000
                                        ARB FAIL CNT = 0.(10)
                                        SEL FAIL CNT = 0.(11)
                                        PARITY ERR CNT = 0.(12)
                                        PHASE ERR CNT = 1.(13)
                                        BUS RESET CNT = 0.(14)
                                        BUS ERROR CNT = 0.(15)
                                        CONTROLLER ERROR CNT = 0.(16)
      SCSI RETRY CNT   00000000
                           0000
                                        ARB RETRY CNT = 0.(17)
                                        SEL RETRY CNT = 0.(18)
                                        BUSY RETRY CNT = 0.(19)
      PHASE QUEUE           0908
                                        2. ELEMENT PHASE QUEUE(20)
                                        _ARBITRATION
                                        _SELECTION

PORT DEPENDENT DATA(21)

      CNTLR INI CMD         02
                                        ATN ASSERTED
      CNTLR MODE            20
                                        PARITY CHECK ENABLED
```

(continued on next page)

Example 17–1 (Cont.) SCSI Bus Phase Error Port Driver Error Log Entry

```
      CNTLR TAR CMD          00
      CNTLR CURR STS         78
                                    C/D ASSERTED
                                    MSG ASSERTED
                                    REQ ASSERTED
                                    BUSY ASSERTED
      CNTLR STATUS           02
                                    ATN ASSERTED
      DMA CNT           00000000
      DMA ADDRESS       00004200
      DMA DIR                01
```

Example 17–2 SCSI Bus Reset Port Driver Error Log Entry

```
                                    READ OPERATION
****************************** ENTRY     209. ********************************
ERROR SEQUENCE 43029.                          LOGGED ON:        SID 0A000005
DATE/TIME 13-SEP-1990 15:03:35.08                              SYS_TYPE 04010102
SCS NODE:                                                       VAX/VMS X5.2-1C

DEVICE ATTENTION  KA420  CPU REV# 6.

SCSI PORT SUB-SYSTEM, UNIT _PKA0:

      ERROR TYPE             09
                                    BUS RESET INITIATED(1)
                                    SUB-ERROR TYPE = 00(X)(2)
      SCSI ID                02
                                    SCSI ID = 2.(3)
      SCSI CMD          CA810208
                             0019
                                    READ(4)
      SCSI MSG               00
                                    COMMAND COMPLETE(5)
      SCSI STATUS            FF
                                    NO STATUS RECEIVED(6)
      PORT ERROR CNT    00000000
                        00000000
                        00000001
                                    BUS BUSY CNT = 0.(7)
                                    UNSOL RESET CNT = 0.(8)
                                    UNSOL INTRPT CNT = 1.(9)
      CONN ERROR CNT    00000000
                        00000000
                        00000000
                        00000001
                        00000001
                        00000000
                        00000000
                                    ARB FAIL CNT = 0.(10)
                                    SEL FAIL CNT = 0.(11)
                                    PARITY ERR CNT = 0.(12)
                                    PHASE ERR CNT = 1.(13)
                                    BUS RESET CNT = 1.(14)
                                    BUS ERROR CNT = 0.(15)
                                    CONTROLLER ERROR CNT = 0.(16)
      SCSI RETRY CNT    00000000
                             0000
                                    ARB RETRY CNT = 0.(17)
                                    SEL RETRY CNT = 0.(18)
                                    BUSY RETRY CNT = 0.(19)
```

(continued on next page)

Example 17–2 (Cont.) SCSI Bus Reset Port Driver Error Log Entry

```
      PHASE QUEUE          0908
                                        2. ELEMENT PHASE QUEUE(20)
                                        _ARBITRATION
                                        _SELECTION

PORT DEPENDENT DATA(21)

      CNTLR INI CMD          00
      CNTLR MODE             00
      CNTLR TAR CMD          00
      CNTLR CURR STS         00
      CNTLR STATUS           08
                                        PHASE MATCH
      DMA CNT          00000000
      DMA ADDRESS      00004200
      DMA DIR                01
                                        READ OPERATION
```

Example 17–3 SCSI Bus Reset Class Driver Error Log Entry

```
****************************** ENTRY     210. ******************************
ERROR SEQUENCE 43030.                          LOGGED ON:        SID 0A000005
DATE/TIME 13-SEP-1990 15:03:35.49                              SYS_TYPE 04010102
SCS NODE:                                                       VAX/VMS X5.2-1C

DEVICE ERROR  KA420  CPU REV# 6.

RZ23 SUB-SYSTEM, UNIT _DKA200:

      HW REVISION      38313630
                                        HW REVISION = 0618(22)
      ERROR TYPE             05
                                        EXTENDED SENSE DATA RECEIVED(23)
      SCSI ID                02
                                        SCSI ID = 2.(24)
      SCSI LUN               00
                                        SCSI LUN = 0.(25)
      SCSI SUBLUN            00
                                        SCSI SUBLUN = 0.(26)
      PORT STATUS      00000001
                                        %SYSTEM-S-NORMAL, NORMAL SUCCESSFUL
                                         COMPLETION(27)
      SCSI CMD         CA810208
                           0019
                                        READ(28)
      SCSI STATUS            02
                                        CHECK CONDITION(29)

EXTENDED SENSE DATA(30)

      EXTENDED SENSE   00060070
                       0C000000
                       00000000
                       00000029
                           0000
                                        UNIT ATTENTION
                                        POWER ON OR RESET OCCURRED
```

(continued on next page)

Example 17–3 (Cont.) SCSI Bus Reset Class Driver Error Log Entry

```
      UCB$B_ERTCNT          00
                                    0. RETRIES REMAINING
      UCB$B_ERTMAX          00
                                    0. RETRIES ALLOWABLE
      ORB$L_OWNER     00010001
                                    OWNER UIC [001,001]
      UCB$L_CHAR      1C4D4008
                                    DIRECTORY STRUCTURED
                                    FILE ORIENTED
                                    SHARABLE
                                    AVAILABLE
                                    MOUNTED
                                    ERROR LOGGING
                                    CAPABLE OF INPUT
                                    CAPABLE OF OUTPUT
                                    RANDOM ACCESS
      UCB$W_STS           0010
                                    ONLINE
      UCB$L_OPCNT     0000104E
                                    4174. QIO'S THIS UNIT

      UCB$W_ERRCNT        0001
                                    1. ERRORS THIS UNIT
      IRP$W_BCNT          3200
                                    TRANSFER SIZE 12800. BYTE(S)
      IRP$W_BOFF          0000
                                    TRANSFER PAGE ALIGNED
      IRP$L_PID       0001000D
                                    REQUESTOR "PID"
      IRP$Q_IOSB      00000000
                      00000000      IOSB, 0. BYTE(S) TRANSFERRED

******************************** ENTRY     211. ********************************
ERROR SEQUENCE 43031.                          LOGGED ON:        SID 0A000005
DATE/TIME 13-SEP-1990 15:04:51.53                              SYS_TYPE 04010102
SCS NODE:                                                       VAX/VMS X5.2-1C

TIME STAMP  KA420  CPU REV# 6.
```

18 Terminal Class and Port Drivers

This chapter describes details of the implementation of the VMS terminal driver. The VMS terminal driver consists of two pieces: the terminal port driver and the terminal class driver. These two pieces of code, when bound together within the unit control block (UCB), form a single device-dependent driver that implements the VMS terminal services.

TTDRIVER.EXE, the VMS terminal class driver, handles the device-independent functions and tasks. For example, it contains code that enables command line editing on many different types of terminals. The port drivers manage those functions and tasks that depend on the device's hardware configuration. For example, the port driver for a particular type of terminal controller performs the actual transmission and reception of characters to and from that terminal controller. The port driver reserves all manipulation and interpretation of those characters to the class driver. Because class driver code supports the functions common to terminal devices, a terminal port driver can contain only that code needed to control a specific interface.

There are several reasons why a new port driver may be required:

- To support a new terminal controller
- To implement a terminal server such as LAT11
- To provide a pseudoterminal

Both class drivers and port drivers adhere to the same rules as other VMS device drivers. They consist of the same routines and tables as standard drivers, and reference the same data structures. However, because a class driver and its port drivers must intercommunicate, they must employ a few additional structures and routines not required for standard VMS device drivers.

The structure of the VMS terminal driver illustrates one specific approach to the class/port concept. The interface described in this chapter relates only to TTDRIVER.EXE and its ports. Programmers of terminal port drivers must adhere to the programming rules presented in this chapter.

Note that there are no supported methods for implementing the class/port design in a non-Digital-supplied device driver. Moreover, the System Generation Utility (SYSGEN) provides no support for alternate class drivers and can only connect port drivers to TTDRIVER.EXE.

The remainder of this chapter describes how the VMS terminal class and port drivers are structured and how they interact. A full description of the functions of the VMS terminal driver appears in the *VMS I/O User's Reference Manual: Part I.*

18.1 Overview

The terminal class driver is the device-independent part of the VMS terminal driver. It contains the driver's function decision table (FDT) routines, start I/O routine, fork process routines, code that implements the features of the VMS terminal services, and the class driver service routines.

The terminal port driver is the device-dependent piece of the VMS terminal driver. It contains the driver prologue table (DPT); data structure initialization; device, unit, and controller initialization routines; port service routines; interrupt service routine; and any additional device-dependent code. Among the port drivers included in VMS are DZDRIVER for the DZ–32 and DZ–11, YCDRIVER for the DMF–32 and DMZ–32, YFDRIVER for the DHU–11 and DHV–11, YIDRIVER for the DMB32, and YEDRIVER for the MicroVAX 2000/3100 and VAXstation 3100 built-in serial lines. (See the *VMS I/O User's Reference Manual: Part I* for a complete list of supported terminal controllers.)

18.2 Data Structures

There are three major data structures that define the communication between the port and the class drivers. These data structures are the UCB, the port driver vector table and the class driver vector table. To reference these structures, a driver must include an invocation of the $TTYDEFS macro (from SYS$LIBRARY:LIB.MLB). The $TTYDEFS macro defines symbolic offsets for the following structures:

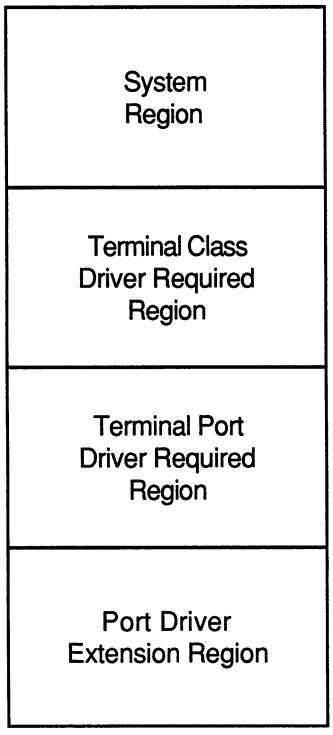
- Unit control block (UCB)
- Terminal UCB extension
- Channel request block (CRB)
- Interrupt dispatch block (IDB)
- Port and class driver vector tables
- Read buffer
- Input stack
- Item list descriptor
- Type-ahead buffer

18.2.1 Terminal UCB

A terminal UCB, as depicted in Figure 18–1, contains four sections: the system section (base UCB), the class driver required section, the port driver required section, and the port extension region.

The system section of the terminal driver UCB contains the pieces of the UCB that are present in all of the UCBs on the system.

Figure 18–1 UCB Structure for Terminal Class/Port Drivers



ZK–6540–GE

The class driver required section of the UCB contains fields that are needed by the class driver. These fields have names of the form UCB$*x*_TT_*fieldname*, where *x* denotes the field size and *fieldname* is the name of the field.

The port driver required section of the UCB contains fields that both the class and port driver must access. These fields have names of the form UCB$*x*_TP_*fieldname*, where *x* denotes the field size and *fieldname* is the name of the field. Although a port driver may not actually use all these fields, their presence is required.

The terminal port extension region is defined by the terminal port driver. It can be any length and contain any context that the port driver needs to perform its duties.

The structures and contents of the UCB and terminal extensions are described in the *VMS Device Support Reference Manual*.

18.2.2 Port Driver Vector Table

The port driver vector table, as depicted in Figure 18–2, is the data structure that allows the terminal class driver to find the port service routines. The vector table contains the address, relative to the beginning of the port driver, of each port service routine. The port driver's controller initialization routine invokes the CLASS_CTRL_INIT macro, as described in Section 18.4.1.1, to relocate this vector table.

The port driver vector table is contained within the port driver itself, usually after the port driver's DPT. The port driver must build its vector table using the $VECINI, $VEC, and $VECEND macros, as described in Section 18.2.4. A field in the UCB, UCB$L_TT_PORT, contains the address of the port driver vector table.

Port and class drivers refer to fields within the port driver vector table using the symbolic offsets represented in Figure 18–2. To use these offsets, they include an invocation of the macro $TTYDEFS (in SYS$LIBRARY:LIB.MLB).

Figure 18–2 Port Driver Vector Table

PORT_STARTIO	0
PORT_DISCONNECT	4
PORT_SET_LINE	8
PORT_DS_SET	12
PORT_XON	16
PORT_XOFF	20
PORT_STOP	24
Reserved	28
PORT_ABORT	32
PORT_RESUME	36
PORT_SET_MODEM	40
Reserved	44
PORT_MAINT	48
PORT_FORKRET	52
PORT_FDT	56

(continued on next page)

Figure 18–2 (Cont.) Port Driver Vector Table

Reserved	60
Reserved	64
Reserved	68
PORT_CANCEL	72

18.2.3 Class Driver Vector Table

The class driver vector table, as depicted in Figure 18–3, contains the address, relative to the beginning of the class driver, of each class service routine. The list is terminated by a longword containing zeros that indicates to the relocation routine where the list ends.

At driver load time, the relative offsets are relocated to actual virtual addresses. The port driver's controller initialization routine invokes the CLASS_CTRL_INIT macro, as described in Section 18.4.1.1, to relocate this vector table. The VMS terminal class driver is loaded by SYSINIT at boot time to allow the console terminal port driver to run.

Figure 18–3 Class Driver Vector Table

CLASS_GETNXT	0
CLASS_PUTNXT	4
CLASS_SETUP_UCB	8
CLASS_DS_TRAN	12
CLASS_DDT	16
CLASS_READERROR	20
CLASS_DISCONNECT	24
CLASS_FORK	28
CLASS_POWERFAIL	32
CLASS_TABLES	36

The class driver vector table is contained within the class driver itself, usually after the class driver's DPT. The class driver builds its vector table using the $VECINI, $VEC, and $VECEND macros, as described in Section 18.2.4. A field in the UCB, UCB$L_TT_CLASS, contains the address of the class driver vector table.

18.2.4 Vector Table Generation Macros

Port drivers use three VMS-supplied macros to build the port driver vector table: $VECINI, $VEC, and $VECEND. Class drivers build the class driver vector table using the same macros. To obtain the definitions for these macros, a driver must invoke the $TTYMACS macro (in SYS$LIBRARY:LIB.MLB). This section briefly discusses the functions of each of these macros. An example of their use appears in Figure 18–4, and a full discussion of their syntax appears in the macro chapter of the *VMS Device Support Reference Manual*.

18.2.4.1 $VECINI Macro

The $VECINI macro creates a vector table and initializes each entry with the address of the driver's null entry point. Subsequent calls to the $VEC macro fill in selected table entries with the addresses of real entry points.

The driver must specify the **drivername** and **null_routine** arguments to the $VECINI macro. The **drivername** argument generally contains a 2-letter prefix to the driver name, such as DZ or YE. The **null_routine** argument contains the address of a routine within the driver (for example DZ$NULL) that contains an RSB. When the class driver attempts to call the port driver at an entry point corresponding to an unsupported function, the port driver's null routine simply returns control to the class driver. The class driver can then proceed to service the error.

18.2.4.2 $VEC Macro

The $VEC macro validates and generates a vector table entry.

Each invocation of the $VEC macro specifies the **entry** argument and the **routine** argument. However, a driver need not supply the address of a routine for each **entry** in the table. The $VEC macro will construct a valid table regardless of how many entries are supplied. The $VEC macro accepts the entry names (minus the PORT_ or CLASS_ prefix) shown in Table 18–1, for port drivers, and Table 18–2, for class drivers. Note that a driver accesses the table using the symbolic offsets indicated in Figures 18–2 and 18–3. The $VECINI macro defines the prefix applied to the entries, which is PORT_ for the port vector table and CLASS_ for the class vector table.

18.2.4.3 $VECEND Macro

The $VECEND macro generates the longword of zeros that terminates the vector table and positions the location counter at label **drivername**$VECEND. It has no required arguments.

18.3 Structure of Port and Class Drivers

Class and port drivers share a similar organization, as seen in Figures 18–4 and 18–5.

The vector table of each follows the driver prologue table (DPT). The driver specifies the address of the vector table in the **vector** argument to the DPTAB macro, which places its offset from the beginning of the DPT in DPT$W_VECTOR.

Following the vector table, and linked to the vectors by invocations of the $VEC macro, is a set of service routines. The balance of the driver includes standard driver routines and tables and driver-specific routines.

Figure 18–4 Port Driver Structure

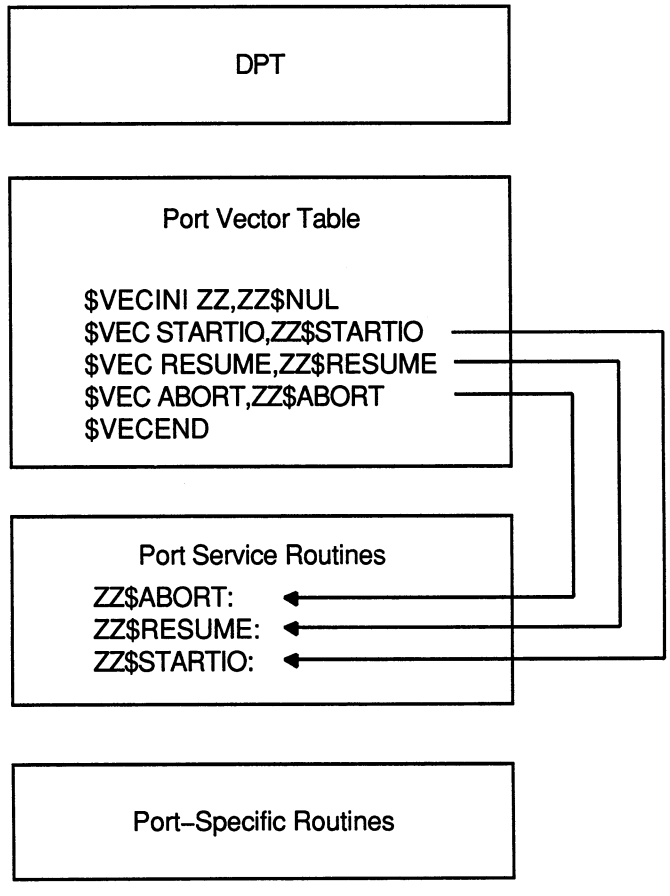
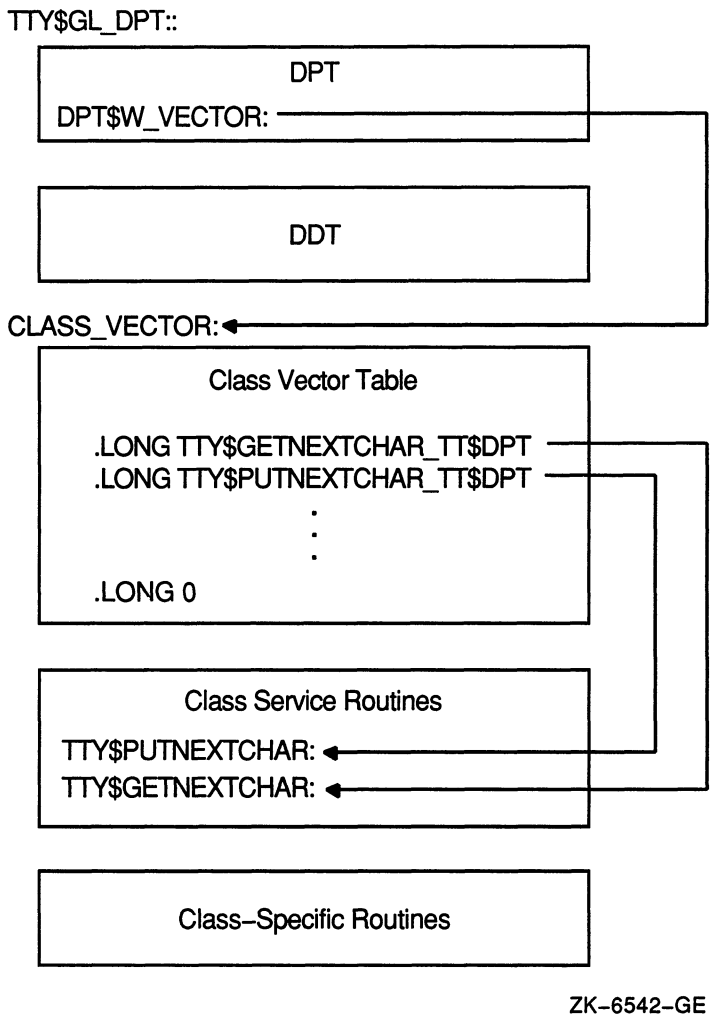


Figure 18–5 Class Driver Structure



18.3.1 Binding Class and Port Drivers

The terminal class and port drivers are bound together to form a single, complete driver in the manner represented in Figure 18–6.

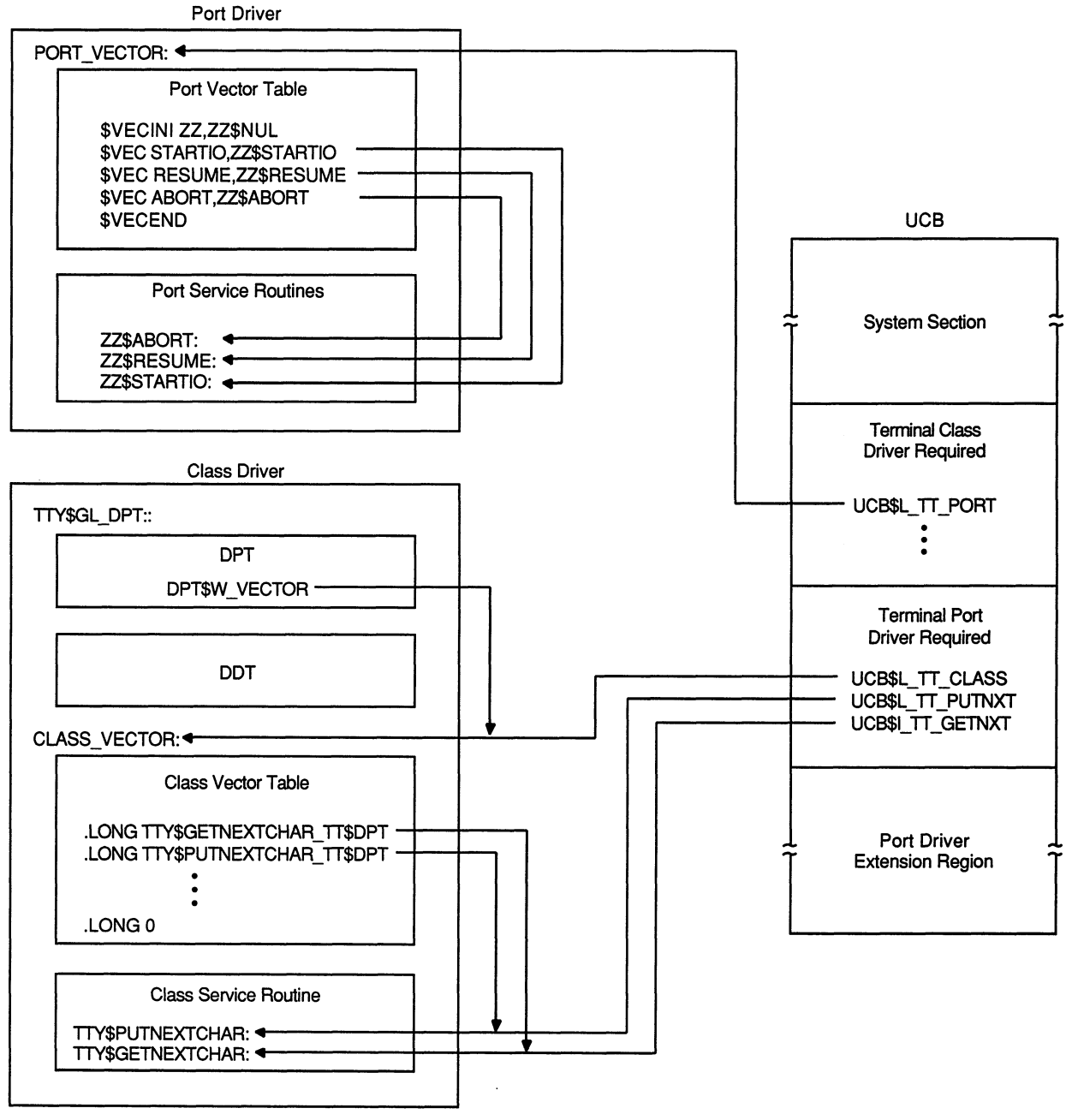
The port driver's unit initialization routine performs the binding process by calling the CLASS_UNIT_INIT macro. The CLASS_UNIT_INIT macro fills in the following UCB fields as indicated:

Field	Contents
UCB$L_TT_CLASS	Terminal class driver's vector table address
UCB$L_TT_PORT	Terminal port driver vector table address
UCB$L_TT_GETNXT	Address of the class driver's get-next-character routine (CLASS_GETNXT)
UCB$L_TT_PUTNXT	Address of the class driver's put-next-character routine (CLASS_PUTNXT)
UCB$L_DDT	Address of the terminal class driver's driver dispatch table

Note that, because the get-next-character and put-next-character routines are the most heavily used class driver routines, their addresses are stored in the UCB. It is therefore possible for code to issue the instruction JSB *@address*(R5) to call either routine (presuming that R5 contains the address of the UCB). To call other routines, the driver must first move the address of the vector table to a general register and issue an instruction of the form JSB *@offset(Rn)*. Although a saving of one instruction does not seem significant, it can save one instruction per character when a driver is receiving data.

When the port driver's unit initialization routine completes the binding, the terminal class and port drivers have become one complete driver, and the device units are ready for I/O.

Figure 18–6 Terminal Class/Port Driver Binding



18.4 Port Driver Routines

When the terminal class driver has completed a segment of device-independent processing of an I/O request, it calls a port routine to complete the device-dependent processing. The port driver contains three types of routines: port startup routines, port initiate routines, and port service routines.

Table 18–1 lists the port driver routines that are part of the class/port interface. This section describes the functions and context of each listed routine.

Table 18–1 Port Driver Routines

Routine	Function
Port Startup Routines	
Controller initialization routine	Resets the controller and relocates the port and class driver vector tables
Unit initialization routine	Sets up each device unit controlled by the driver
Port Initiate Routines	
PORT_DISCONNECT	Notifies the port driver of the last deassign for the UCB
PORT_DS_SET	Outputs modem signals to a specified unit
PORT_FDT	Performs FDT processing for device-specific function modifiers
PORT_FORKRET	Return address in the port driver to which CLASS_FORK transfers control when servicing the port driver's request for a fork process
PORT_MAINT	Services $QIO requests for IO$_SETMODE function with the IO$M_MAINT modifier
PORT_SET_LINE	Changes terminal line parameters
PORT_SET_MODEM	Informs the port that a line has been enabled for modem signal input transitions
PORT_STARTIO	Starts output on an inactive line

(continued on next page)

Table 18–1 (Cont.) Port Driver Routines

Routine	Function
Port Service Routines	
PORT_ABORT	Aborts any currently active output
PORT_CANCEL	Cancels internally queued operations in response to a $CANCEL request
PORT_RESUME	Resumes any previously stopped output
PORT_STOP	Halts the output data stream
PORT_XOFF	Takes steps to halt an input data stream that is approaching its limit
PORT_XON	Resumes the acceptance of input data

18.4.1 Port Startup Routines

Port startup routines include the port driver's controller and unit initialization routines. Note that, although these routines are not included in the port vector table, they must make calls to several class routines. They additionally fill the role of the equivalent initialization routines in a standard device driver, as discussed in Section 11.1.

18.4.1.1 Controller Initialization Routine

The controller initialization routine is responsible for resetting the controller and relocating the port and class driver's vector tables. To perform the last-mentioned task, the routine should invoke the CLASS_CTRL_INIT macro, supplying the symbolic name of the driver prologue table (for instance, DZ$DPT) in the **dpt** argument and the address of the port driver's vector table in the **vector** argument. To use the CLASS_CTRL_INIT macro, the driver must include an invocation of the $TTYMACS definition macro (from SYS$LIBRARY:LIB.MLB).

18.4.1.2 Unit Initialization Routine

The unit initialization routine is responsible for setting up each individual device unit. The activities of a standard unit initialization routine include loading certain locations in the UCB with controller-specific data, preparing the hardware for input and output, and taking any action necessary to service a power failure.

The unit initialization routine of a terminal port driver must additionally perform the following tasks:

1 Invoke the CLASS_UNIT_INIT macro to generate the common code that must be executed by all terminal port driver unit initialization routines. This code includes the logic that binds the class and port drivers in the manner discussed in Section 18.3.1. Before it invokes the CLASS_UNIT_INIT macro, the unit initialization routine must place the address of the port driver vector table in R0.

To use the CLASS_UNIT_INIT macro, the driver must include an invocation of the $TTYMACS definition macro (from SYS$LIBRARY:LIB.MLB).

2 Call the class service routine, CLASS_SETUP_UCB, to allow the class driver to reset fields in the UCB.

3 Call the class service routine, CLASS_SET_LINE, to allow the class driver to reset the speed, parity, and the device-dependent bits, if necessary.

4 If the line can run modem protocol, call the class service routine, CLASS_DS_TRANS, with the transition type MODEM$C_INIT in R1. Note that, to use modem symbols, the port driver must invoke the $TTYMDMDEF macro (from SYS$LIBRARY:LIB.MLB).

5 When a power failure occurs (UCB$V_POWER set in UCB$W_STS), call the class service routine, CLASS_POWERFAIL.

6 Perform other hardware-specific functions.

18.4.2 Port Initiate Routines

The terminal class driver calls port initiate routines when it must initiate device activity and the port driver is not active. Port initiate routines can issue callbacks to the class driver.

A call to a port initiate routine uses the following instruction format:

```
MOVL    UCB$L_TT_PORT(R5),R0    ;get pointer to port vector table
JSB     @PORT_DISCONNECT(R0)    ;call port disconnect routine
```

Note that a port initiate routine must preserve the contents of all registers.

18.4.2.1 PORT_DISCONNECT

A call to the PORT_DISCONNECT routine indicates that there are no longer channels associated with the device, thus notifying the port driver of the last deassignment for the device's UCB. If the delete bit (UCB$V_DELMBX) is set in UCB$W_DEVSTS, VMS will delete the UCB.

Note: As long as the device name is known to the system, broadcasts and assign channel requests may occur on this device. (Broadcasts, however, will not occur if the DEV$V_NET bit is set in UCB$L_DEVCHAR.)

Input to the PORT_DISCONNECT routine is as follows:

R0 Flags. If bit 0 is set, the user requested that the UCB not be deleted (NOHANGUP).

R5 Address of UCB.

18.4.2.2 PORT_DS_SET

The PORT_DS_SET routine sends modem signals to the specified unit. Masks representing modem signals are defined in $TTDEF (in SYS$LIBRARY:STARLET.MLB). They include the following:

TT$M_DS_CARRIER	Data channel received line signal detector
TT$M_DS_CTS	Clear to send
TT$M_DS_DSR	Data set ready

TT$M_DS_DTR	Data terminal ready
TT$M_DS_RING	Calling indicator
TT$M_DS_RTS	Request to send
TT$M_DS_SECREC	Secondary receive
TT$M_DS_SECTX	Secondary transmit

See the *VMS I/O User's Reference Manual: Part I* for an explanation of modem protocol.

Input to the PORT_DS_SET routine is as follows:

R2 Low byte indicates signals to be activated; high byte indicates signals to be deactivated.

R5 Address of UCB.

18.4.2.3 PORT_FDT

The terminal class driver calls the PORT_FDT routine when servicing a $QIO request for an IO$_TTY_PORT function. The PORT_FDT routine performs whatever tasks the class driver's FDT routine would normally do to service the request. These tasks include checking the function-dependent parameters (**p1** through **p6**), verifying access to buffers, and terminating with a call to EXE$QIORETURN, EXE$ABORTIO, or EXE$FINISHIO.

The PORT_FDT routine thus allows a port driver to implement support for device-specific function modifiers without requiring an extension to the class/port interface.

If there is no PORT_FDT routine, control will pass to the port driver's null routine, which returns control to the class driver. The VMS terminal class driver, TTDRIVER.EXE, thereupon issues an illegal I/O function error.

Input to the PORT_FDT routine is as follows:

R3	Address of IRP
R4	Address of current PCB
R5	Address of UCB
R6	Address of CCB
R7	Bit number of the I/O function code
00(AP)	Address of the first function-dependent QIO parameter (**p1**)

The routine destroys the contents of R2.

Note that a port driver must set TTY$V_PC_PORTFDT in UCB$W_TT_PRTCTL if it contains a PORT_FDT routine.

18.4.2.4 PORT_FORKRET

The terminal class driver's service routine, CLASS_FORK, returns control to the port driver's PORT_FORKRET entry point after servicing the port driver's request to create a fork process. (See the description of CLASS_FORK in Section 18.5.4.) The only context returned from the servicing of the fork request is the address of the UCB in R5.

The terminal class driver issues a a JMP instruction (rather than a JSB instruction) to the PORT_FORKRET routine.

18.4.2.5 PORT_MAINT

The class driver calls the PORT_MAINT routine whenever a $QIO request is issued for an IO$_SETMODE function with the IO$M_MAINT modifier. The *VMS I/O User's Reference Manual: Part I* lists all possible maintenance functions; each port driver must decide which of these functions it must support.

Input to the PORT_MAINT routine is as follows:

R5	Address of UCB
UCB$B_TT_MAINT	Parameters to the IO$M_MAINT function

18.4.2.6 PORT_SET_LINE

The PORT_SET_LINE routine changes terminal line parameters. The terminal class driver calls the PORT_SET_LINE routine whenever any terminal characteristic in UCB$L_DEVDEPEND or UCB$L_DEVDEPND2 is changed. It also calls this routine when speed, parity, and the enabling or disabling of DMA and automatic flow control are affected.

The PORT_SET_LINE routine is the only port routine that is allowed to write the fields UCB$L_DEVDEPEND and UCB$L_DEVDEPND2.

Input to the PORT_SET_LINE routine is as follows:

R5	Address of UCB.
UCB$B_TT_MAINT	Parameters to the IO$M_MAINT function.
UCB$B_TT_PARITY	Parity, stop bits, and frame size.
UCB$W_TT_SPEED	Low byte indicates transmit speed; high byte indicates receive speed or is zero.
UCB$W_TT_PRTCTL	DMA enable flag (TTY$V_PC_DMAENA) and auto XOFF enable flag (TTY$V_PC_XOFENA).
UCB$L_DEVDEPEND	First longword for device-dependent status.
UCB$L_DEVDEPND2	Second longword for device-dependent status.

The PORT_SET_LINE routine can safely destroy the contents of R4.

18.4.2.7 PORT_SET_MODEM

A call to the PORT_SET_MODEM routine informs the port that the line has been enabled for modem signal input transitions. A port implementing modem functions must ensure that the hardware is ready to detect changes in input modem signals. When hardware does not provide this capability (as, for instance, the DZ11 terminal controller does not), the VMS terminal class/port interface implements the equivalent capability by using timer-based polling.

At the time of the call, R5 contains the address of the UCB.

18.4.2.8 PORT_STARTIO

The terminal class driver calls the PORT_STARTIO routine to start output on a line that is currently inactive. The PORT_STARTIO routine is always called with either a character or a burst of data and it is never called unless the line is idle (UCB$V_INT is clear in UCB$W_STS).

The UCB$V_INT bit functions as an interlock, signifying that the port output logic is busy. The class driver always sets UCB$V_INT when it calls PORT_STARTIO. If the port requests that timers be set up (TTY$V_PC_NOTIME clear in UCB$W_TT_PRTCTL), then the class driver calculates and creates an output timer for the burst or character and sets UCB$V_TIM in UCB$L_STS.

Input to the PORT_STARTIO routine is as follows:

R0	Address of the port driver's vector table (UCB$L_TT_PORT)
R2	Address of the terminal state quadword (UCB$L_TT_STATE1)
R3	Character to be output (if UCB$B_TT_OUTYPE is 1)
R5	Address of UCB
UCB$B_TT_OUTYPE	Zero if there is no character to be output; 1 if there is one character to be output; and a negative value if there is a burst to be output
UCB$L_TT_OUTADR	Address of burst to output (if UCB$B_TT_OUTYPE is negative)
UCB$W_TT_OUTLEN	Length of burst (if UCB$B_TT_OUTYPE is negative)

18.4.3 Port Service Routines

The terminal class driver can call port service routines at any time.

Note: Because they must consist of reentrant code, port service routines cannot issue callbacks to the class driver.

A call to a port service routine uses the following instruction format:

```
MOVL    UCB$L_TT_PORT(R5),R0    ;get pointer to port vector table
JSB     @PORT_ABORT(R0)         ;call port abort routine
```

Note that a port service routine must preserve the contents of all registers.

18.4.3.1 PORT_ABORT

The terminal class driver calls the PORT_ABORT routine to abort any currently active output activity: for instance, the last burst of output sent to the port. The PORT_ABORT routine invalidates the contents of the address stored in UCB$L_TT_OUTADR.

At the time of the call, R5 contains the address of the UCB.

18.4.3.2 PORT_CANCEL

The terminal class driver calls the PORT_CANCEL routine in servicing a $CANCEL, $DASSGN, or $DALLOC request. The PORT_CANCEL routine cancels any internally queued operations for the port. Most commonly, a call is issued to this routine when a request to establish an outgoing connection has been stalled because the port is busy.

Input to the PORT_CANCEL routine is as follows:

R2	Address of the terminal state quadword (UCB$L_STATE1)
R3	Address of IRP
R4	Address of current PCB
R5	Address of UCB
R6	Channel index number
R8	Reason for cancellation, one of the following:
	CAN$C_CANCEL — Called by $CANCEL system service
	CAN$C_DASSGN — Called by $DASSGN or $DALLOC system service
R9	Address of logical UCB (UCB$L_TT_LOGUCB)

18.4.3.3 PORT_RESUME

The terminal class driver calls the PORT_RESUME routine to resume any previously stopped output. The port must be prepared for this routine to be called at any time (whether output is currently active or has previously been stopped). The PORT_RESUME routine should always ensure that the port hardware is enabled for output.

At the time of the call, R5 contains the address of the UCB.

18.4.3.4 PORT_STOP

The PORT_STOP routine halts the output data stream. The terminal class driver normally calls this routine in response to input flow control. When called, the PORT_STOP routine should stop the data stream as soon as possible.

At the time of the call, R5 contains the address of the UCB.

18.4.3.5 PORT_XOFF

The terminal class driver calls the PORT_XOFF routine when it is approaching or has reached its input limit. The PORT_XOFF routine takes steps to stop the input data stream. For character-oriented controllers, it commands the port to insert the flow control character in the output data stream as soon as possible.

Input to the PORT_XOFF routine is as follows:

R3	Flow control character to be inserted in the input data stream
R5	Address of UCB
UCB$L_STS	UCB$V_INT may or may not be set

If the port hardware requires it, the PORT_XOFF routine sets UCB$V_INT in UCB$L_STS.

18.4.3.6 PORT_XON

The terminal class driver calls the PORT_XON routine when it has cleared its input path and is ready to accept data. For character-oriented controllers, the PORT_XON routine commands the port to insert the flow control character in the input data stream.

Input to the PORT_XON routine is as follows:

R3	Flow control character to be inserted in the input data stream
R5	Address of UCB
UCB$L_STS	UCB$V_INT may or may not be set

If the port hardware requires it, the PORT_XON routine sets UCB$V_INT in UCB$L_STS.

18.4.3.7 Port Interrupt Service Routines

A terminal port driver must contain code to service receiver interrupts and transmitter interrupts. The exact form of a device interrupt associated with the port driver is device dependent. For multiple-line interfaces, the port driver must also determine which line is requesting the interrupt and move its UCB address into R5.

To service *receiver interrupts*, the port driver obtains a character from the port, together with hardware error flags that signal a parity, overrun, or frame error in the transaction. It proceeds as follows:

- If an error has been detected, the port driver passes the character and error flags to the CLASS_READERROR service routine for processing (for instance, autobaud detection).
- If no error has been detected, the port driver passes the character to the CLASS_PUTNXT service routine.

If either of these service routines returns characters that must be echoed and the line is currently inactive, the port driver must start output. Before dismissing the interrupt, the port driver for a controller with multiple lines should check all lines for pending transactions and empty the silo.

To service *transmitter interrupts*, the port driver first records any reported errors. It then proceeds as follows:

- If TTY$V_TP_ABORT is set in UCB$B_TP_STAT, the port driver calls the PORT_ABORT service routine to terminate the transaction.
- If TTY$V_TP_ABORT is not set, the port driver sets up the next output sequence according to the following priority:

Transaction	Use
Preempt	Normally used to send an XON or XOFF character
Hold	Normally used for single-character output
Burst	Used for multiple-character (DMA) output

Note that this action can result in an XON or XOFF character appearing in the middle of an escape sequence.

18.5 Class Driver Routines

Table 18–2 lists the class driver routines that are part of the class/port interface. This section describes the functions and context of each listed routine.

A call to a class service routine uses the following instruction format:

```
MOVL    UCB$L_TT_CLASS(R5),R0  ;get pointer to class vector table
JSB     @CLASS_DISCONNECT(R0)  ;call class disconnect routine
```

Table 18–2 Class Driver Routines

Routine	Function
CLASS_DDT	Pointer to the driver dispatch table
CLASS_DISCONNECT	Disconnects a process from a terminal on a nonmodem line
CLASS_DS_TRANS	Manages data set transitions
CLASS_FORK	Services a port driver's request to create a fork process
CLASS_GETNXT	Delivers to the port driver the next character or burst to be output
CLASS_PUTNXT	Obtains input characters from the port driver
CLASS_SETUP_UCB	Initializes the UCB
CLASS_POWERFAIL	Services a power failure
CLASS_READERROR	Services a parity, data overrun, or framing error on a terminal line

18.5.1 CLASS_DDT

This entry in the class driver vector table points to the driver dispatch table (DDT). The CLASS_UNIT_INIT macro uses the CLASS_DDT entry point when moving the address of the DDT into the UCB.

18.5.2 CLASS_DISCONNECT

A port driver calls the CLASS_DISCONNECT routine to indicate to the terminal class driver that the terminal is no longer connected to the system. This is the preferred way of disconnecting a process from a terminal on a nonmodem line.

At the time of the call, R5 must contain the address of the UCB. The CLASS_DISCONNECT routine destroys the contents of R4.

18.5.3 CLASS_DS_TRANS

This CLASS_DS_TRANS routine manages data set state transitions. The port driver's unit initialization routine must call this routine with the transition type MODEM$C_INIT in R1 if the unit is capable of having data set transitions. (Note that, to use modem symbols, the port driver must invoke the $TTYMDMDEF data structure definition macro (from SYS$LIBRARY:LIB.MLB).

Input to the CLASS_DS_TRANS routine is as follows:

R1	Transition type, one of the following:	
	MODEM$C_INIT	Initialize modem control
	MODEM$C_INIT_NORESET	Start modem protocol, but do not initialize signals
	MODEM$C_SHUTDWN	Shut down the line and disconnect the process
	MODEM$C_SHUTDWN_NOHANGUP	Stop modem protocol but do not stop the signals
	MODEM$C_DATASET	Data set signal changes
R2	New receive modem mask (if MODEM$C_DATASET is specified in R1)	
R5	Address of UCB	

The CLASS_DS_TRANS routine destroys the contents of R0 through R4.

18.5.4 CLASS_FORK

A port driver calls the CLASS_FORK routine to create a driver fork process that uses the UCB fork block. The port driver must never initiate a fork directly—it must always call this routine.

The CLASS_FORK routine sets up the fork block in the UCB and performs the other tasks necessary to store context in the fork block, insert it in a processor-specific fork queue, and suspend driver processing. When the fork has taken place, the class driver calls the port driver at its PORT_FORKRET entry point.

At the time of the call, R5 must contain the address of the UCB. The CLASS_FORK routine destroys the contents of R4.

18.5.5 CLASS_GETNXT

The port driver calls the CLASS_GETNXT routine whenever it has completed the current character or burst to obtain the next characters to be output on the unit. If CLASS_GETNXT returns data to the port driver, a timer is set up (unless explicitly disabled) and the interrupt expected bit is set.

At the time of the call, R5 must contain the address of the UCB. Output from the CLASS_GETNXT routine includes the following:

R1	Destroyed
R2	Number of characters (if R3 contains an address)
R3	Character to be output (if UCB$B_TT_OUTYPE is positive); address of characters to be output (if UCB$B_TT_OUTYPE is negative); or no character (if UCB$B_TT_OUTYPE is zero)
R4	Destroyed
R5	Address of UCB
R6 through R11	Destroyed
UCB$B_TT_OUTYPE	Zero if there is no character to be output; 1 if there is one character to be output; and a negative value if there is a burst to be output
UCB$L_TT_OUTADR	Address of burst to output (if UCB$B_TT_OUTYPE is negative)
UCB$W_TT_OUTLEN	Length of burst (if UCB$B_TT_OUTYPE is negative)

18.5.6 CLASS_PUTNXT

The port driver calls the CLASS_PUTNXT routine to pass input characters to the terminal class driver. The CLASS_PUTNXT routine filters characters received from nonpassall units for immediate control sequences. If a slave mode unit (that is, generating no unsolicited input) does not have a read outstanding, the CLASS_PUTNXT routine ignores the input characters, after performing the control-character filtering.

If the input characters are to be echoed to the terminal, CLASS_PUTNXT calls CLASS_GETNXT to notify the port driver.

The CLASS_PUTNXT routine may or may not return output data to the port driver, depending upon the setting of the interrupt-expected bit (UCB$V_INT) in UCB$L_STS. If this bit is set, CLASS_PUTNXT does not return data. If it does return data, the terminal port driver should assume that more data may follow, and call CLASS_GETNXT after outputting the returned data.

Input to the CLASS_PUTNXT routine is as follows:

R3	Input character
R5	Address of UCB

Output from the CLASS_PUTNXT routine is as follows:

R1	Destroyed
R2	Number of characters (if R3 contains an address)
R3	Character to be output (if UCB$B_TT_OUTYPE is positive); address of characters to be output (if UCB$B_TT_OUTYPE is negative); or no character (if UCB$B_TT_OUTYPE is zero)

R4	Destroyed
R5	Address of UCB
R6 through R11	Destroyed
UCB$B_TT_OUTYPE	Zero if there is no character to be output; one if there is one character to be output; and a negative value if there is a burst to be output
UCB$L_TT_OUTADR	Address of burst to output (if UCB$B_TT_OUTYPE is negative)
UCB$W_TT_OUTLEN	Length of burst (if UCB$B_TT_OUTYPE is negative)

18.5.7 CLASS_SETUP_UCB

A port driver's unit initialization routine calls CLASS_SETUP_UCB when it is invoked at system startup and power failure.

The CLASS_SETUP_UCB routine initializes the unit's fork block; write queue (UCB$L_TT_WFLINK); break, passall, and DMA device characteristics; and read timed out dispatch field (UCB$L_TT_RTIMOU).

In addition, it initializes several UCB fields as follows:

UCB$L_TT_LOGUCB	Address of UCB
UCB$L_DEVCHAR	DEV$V_AVL set
UCB$L_DEVCHAR2	DEV$V_RED cleared
UCB$W_TT_CURSOR	1
UCB$W_TT_HOLD	Cleared
UCB$W_TT_SPEED	UCB$W_TT_DESPEE
UCB$B_TT_PARITY	UCB$B_TT_DEPARI
UCB$B_DEVTYPE	UCB$B_TT_DETYPE

At the time of the call, R5 must contain the address of the UCB.

18.5.8 CLASS_POWERFAIL

A port driver's unit initialization routine calls the CLASS_POWERFAIL routine when it detects a power failure.

At the time of the call, R5 must contain the address of the UCB.

Output from the CLASS_POWERFAIL routine includes the following:

UCB$W_STS	UCB$V_INT cleared; UCB$V_TIM set
UCB$L_DUETIM	Cleared

18.5.9 CLASS_READERROR

A port driver calls CLASS_READERROR when it detects a parity, data overrun, or framing error on the terminal line. CLASS_READERROR completes the read operation with error status if a read is active, or simply returns if no read is active.

Input to the CLASS_READERROR routine is as follows:

R3 Character and flags. The following flags are defined:

- Bit 12 Parity error on the given character
- Bit 13 Framing error on the given character
- Bit 14 Data overrun

R5 Address of UCB.

Output from the CLASS_READERROR routine is as follows:

R0 through R3	Destroyed
UCB$B_TT_OUTYPE	Zero if there is no character to be output; 1 if there is one character to be output; and a negative value if there is a burst to be output

19 Mapping to I/O Space and the Connect-to-Interrupt Facility

Programs written in VAX MACRO can interface with the I/O subsystem by using VMS RMS, by using the Queue I/O Request ($QIO) system service, or by mapping to I/O address space and connecting to a device interrupt vector. Programs written in a high-level language can interface with the I/O subsystem using the same methods as a VAX MACRO program, or they can issue the I/O statements specific to that language. In the latter case, the program interfaces with the I/O subsystem by means of the VAX Common Run-Time Procedure Library.

A user program can interface with the I/O subsystem at one of several levels, depending on its requirements. At each level, the user program makes trade-offs between ease of use and execution speed. As a general rule, the closer to the VMS executive that a user program interfaces, the less overhead is involved in the I/O operation. The connect-to-interrupt capability offers the least overhead.

A process with suitable privileges can connect to a device interrupt vector or map the system's I/O address space into process virtual address space or both. Connecting to a device interrupt vector allows your process to respond to interrupts from the device with minimal overhead. Mapping system I/O address space allows your process to access device registers from the main program or from an AST procedure.

A process normally uses these features for devices that do not have VMS drivers. These devices must not be direct memory access (DMA) devices, and they must be attached to the UNIBUS or Q22 bus. Examples of such devices are the AXV11–C and the KW11–P.

19.1 I/O Address Space

In a VAX system, I/O address space is assigned physical address locations of 20000000_{16} and higher ($F20000_{16}$ and higher for VAX–11/730 and VAX–11/780 systems). I/O address space contains device registers that a driver or user process can read and write to control a device. Each device controller has an associated control and status register (CSR) in I/O address space. Device registers for each device are located at an offset from the device's CSR.

Macros of the format $IO*xxx*DEF (where *xxx* represents a specific VAX system), contained in SYS$LIBRARY:LIB.MLB, define symbols describing the layout of I/O address space. Table 19–1 describes these macros and the symbols they define for each VAX system.

Table 19–1 Symbols Defined by the $IO*xxx*DEF Macros

Symbols	Meaning	Value (Hex)
For VAX 6000-2xx/6000-3xx Series defined by macro $IO9CCDEF		
IO9CC$AL_IOBASE	Start of I/O address space	20000000
IO9CC$C_BIWINDOW	Offset to node 0 window space	400000
IO9CC$C_BIWSIZ	Size of window space	40000
IO9CC$C_PERXBI	Size of adapter address space	2000000
For VAX 6000-4xx Series defined by macro $IO9RRDEF		
IO9RR$AL_IOBASE	Start of I/O address space	20000000
IO9RR$C_BIWINDOW	Offset to node 0 window space	400000
IO9RR$C_BIWSIZ	Size of window space	40000
IO9RR$C_PERXBI	Size of adapter address space	2000000
For VAX 85*x*0/8700/88*x*0 defined by macro $IO8NNDEF		
IO8NN$AL_NBIB_0	Start of I/O address space for VAXBI 0	20000000
IO8NN$AL_NBIB_1	Start of I/O address space for VAXBI 1	22000000
IO8NN$AL_NBIB_2	Start of I/O address space for VAXBI 2	24000000
IO8NN$AL_NBIB_3	Start of I/O address space for VAXBI 3	26000000
IO8NN$AL_NBIB_4	Start of I/O address space for VAXBI 4	28000000
IO8NN$AL_NBIB_5	Start of I/O address space for VAXBI 5	2A000000
IO8NN$AL_NODESP	Offset to node 0 window space	400000
IO8NN$AL_NDSPER	Size of window space	40000
For VAX 82*x*0/83*x*0 defined by macro $IO8SSDEF		
IO8SS$AL_NODESP	Address of node 0 window space	20400000
IO8SS$AL_NDSPER	Size of window space	40000
For VAX 8600/8650 defined by macro $IO790DEF		
IO790$AL_IOA0	Start of I/O address space for SBI0	20000000
IO790$AL_IOA1	Start of I/O address space for SBI1	22000000
IO790$AL_UB0SP	Offset to start of adapter address space for first UNIBUS	24000000
For VAX–11/780 and VAX–11/785 defined by macro $IO780DEF		
IO780$AL_IOBASE	Start of I/O address space	20000000
IO780$AL_UB0SP	Start of adapter address space for first UNIBUS	20100000

(continued on next page)

Table 19–1 (Cont.) Symbols Defined by the $IO*xxx*DEF Macros

Symbols	Meaning	Value (Hex)
For VAX–11/750 defined by macro $IO750DEF[1]		
IO750$AL_IOBASE	Start of I/O address space	F20000
IO750$AL_UBBASE	Start of UBA0 adapter register space	F30000
IO750$AL_MBBASE	Start of MBA0 adapter register space	F28000
IO750$AL_UB0SP	Start of adapter address space for first UNIBUS	FC0000
For VAX–11/730 defined by macro $IO730DEF		
IO730$AL_IOBASE	Start of I/O address space	F20000
IO730$AL_UB0SP	Start of adapter address space for UNIBUS	FC0000
MicroVAX 3400 Series defined by $IO640DEF macro		
IO640$AL_QB0SP	Start of adapter address space for Q22bus	20000000
MicroVAX 3600/3900 Series defined by $IO650DEF macro		
IO650$AL_QB0SP	Start of adapter address space for Q22 bus	20000000
VAX 4000 Series defined by $IO670DEF macro		
IO670$AL_QB0SP	Start of adapter address space for Q22 bus	20000000
MicroVAX II defined by $IOUV2DEF macro		
IOUV2$AL_QB0SP	Start of adapter address space for Q22 bus	20000000

[1]The VAX–11/750 system has fixed MASSBUS adapters (UBBASE, MBBASE) in contrast to the VAX–11/780 system, which has floating MASSBUS adapters, and the VAX–11/730, which does not have MASSBUS adapters.

The number of registers and their locations varies from device to device. The *PDP–11 Peripherals Handbook* provides the necessary information for Digital-supplied devices. The *VAX Hardware Handbook* contains information about the layout of I/O address space.

From the symbols defined by the macros described in Table 19–1, you can derive the starting physical addresses of UNIBUS or Q22 bus adapter address space for the various VAX systems. Table 19–2 lists the starting physical addresses for UNIBUS adapters on the VAX 8600/8650, VAX–11 /780, VAX–11/785, VAX–11/750, and VAX–11/730 systems, as well as the starting physical addresses for MicroVAX/Q22 bus interface address space.

Note: To access UNIBUS device CSRs you must add $3E000_{16}$ to the addresses listed in Table 19–2. This operation is not necessary when you use the values supplied for MicroVAX/Q22 bus systems.

For VAX 85*x*0/8700/88*x*0, VAX 82*x*0/83*x*0, and VAX 6000-series systems, Example 19–1 illustrates the calculations that are necessary to determine the location of the adapter address space for a given UNIBUS adapter (DWBUA or DWMBA) on a VAXBI bus. For additional information on the layout of VAXBI I/O address space, see the discussion and illustrations in Section 16.2.

Table 19–2 UNIBUS and Q22 Bus Adapter Address Space

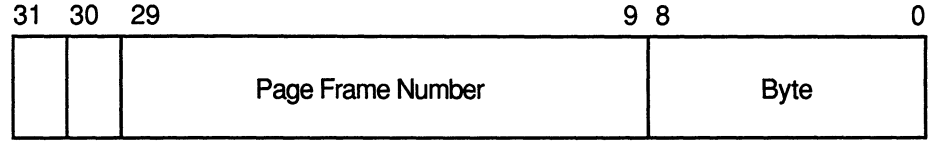
UNIBUS Adapter Number	VAX–11/730	VAX–11/750	VAX–11/780 VAX–11/785	MicroVAX Systems	VAX 8600 SBI0/SBI1 VAX 8650 SBI0/SBI1
0	00FC0000	00FC0000	20100000	20000000	20100000/22100000
1	–	00F80000	20140000	–	20140000/22140000
2	–	–	20180000	–	20180000/22180000
3	–	–	201C0000	–	201C0000/221C0000

Example 19–1 Locating the Adapter Address Space of a UNIBUS Adapter on a VAXBI Bus

```
For VAX 85x0/8700/88x0:

    IO8NN$AL_NBIB_n                          ;Start of I/O address space for given
                                             ;VAXBI bus
  + IO8NN$AL_NODESP                          ;Offset to window space
  + IO8NN$AL_NDSPER * VAXBI-node-ID-of-DWBUA ;Offset to given VAXBI node window space
  + UNIBUS-address-of-device-CSR             ;Offset to device CSR

For VAX 82x0/83x0:

    IO8SS$AL_NODESP                          ;Start of window space
  + IO8SS$AL_NDSPER * VAXBI-node-ID-of-DWBUA ;Offset to given VAXBI node window space
  + UNIBUS-address-of-device-CSR             ;Offset to device CSR

For VAX 6000 series:

    IO9CC$AL_IOBASE                          ;Start of I/O address space
  + IO9CC$_BIWINDOW                          ;Offset to node 0 window space
  + XMI-node-ID-of-XBI * IO9CC$C_PERXBI      ;Offset to given VAXBI bus
  + IO9CC$C_BIWSIZ * VAXBI-node-ID-of-DWMBA  ;Offset to given VAXBI node window space
  + UNIBUS-address-of-device-CSR             ;Offset to device CSR
```

For most VAX processors, the **page frame number** (PFN) of a physical page in memory is contained in bits 9 through 29 of its physical address (see Figure 19–1). Bit 29 of the address is clear to indicate a physical memory address and set to indicate an address in I/O address space. Bits 0 through 8 specify the byte address within the page.

Figure 19–1 Format of a Physical Address



ZK–4845–GE

19.2 PFN Mapping

For a process to gain access from an outer access mode to I/O address space or to any page of physical memory, it must map that page into its virtual address space. When a VMS process maps a page by specifying its page frame number, it completely bypasses VMS memory management and creates its own window to the page. As a result, the protection functions that VMS normally performs are not performed for PFN mapping:

- No checks are performed to ensure that no other VMS processes are mapped to the page and modifying it.
- No reference count is maintained. A process can delete a global section mapped by page frame numbers when other processes are still using it; this is not the case for other types of global sections.

Modifying pages mapped by page frame numbers can have unpredictable results and can adversely affect system operation, especially if the operating system is also using these pages or accessing devices whose registers are in the same pages. Because PFN-mapped pages are not inherently protected from such modification, a process must have the PFNMAP privilege to use this capability.

When used for PFN mapping, the Create and Map Section ($CRMPSC) system service designates the specified page(s) as a global or private section and maps the section into the requesting process's virtual address space. The pages can be located anywhere in the VAX system's local memory, in MA780 memory (if a multiport memory unit is connected to the system), or in I/O address space.

The format and conventions for PFN mapping (that is, mapping a physical page frame section) are similar to those for mapping a disk file section. The $CRMPSC system service has the following general formats:

VAX MACRO Format

```
$CRMPSC  [inadr] [,retadr] [,acmode] [,flags] [,gsdnam] [,ident] -
         [,relpag] [,chan] [,pagcnt] [,vbn] [,prot] [,pfc]
```

High-Level Language Format

SYS$CRMPSC ([inadr] [,retadr] [,acmode] [,flags] [,gsdnam] [,ident]
[,relpag] [,chan] [,pagcnt] [,vbn] [,prot] [,pfc])

The **relpag**, **chan**, and **pfc** arguments are not applicable to mapping by page frame number. The **inadr**, **retadr**, **acmode**, **gsdnam**, **ident**, and **prot** arguments have the same functions regardless of whether you specify page frame number mapping. The *VMS System Services Reference Manual* further describes these arguments.

The following arguments can have values specific to PFN mapping.

Arguments

[flags]
Mask defining the section type and characteristics. This mask is the logical OR of the flag bits you want to set. The $SECDEF macro defines symbolic names for the flag bits in the mask. The SEC$M_PFNMAP flag bit must be set to indicate mapping by page frame number. The SEC$M_PFNMAP flag setting identifies the memory for the section as starting at the page frame number specified in the **vbn** argument and extending for the number of pages specified in the **pagcnt** argument.

If appropriate, the following flags can also be set:

Flag	Description
SEC$M_GBL	Pages form a global section. The default is private section.
SEC$M_EXPREG	Pages are mapped into the first available space. By default, pages are mapped into the range specified by the **inadr** argument.
SEC$M_WRT	Pages form a read/write section. By default, pages form a read-only section.
SEC$M_PERM	Pages are permanent. By default, pages are temporary.
SEC$M_SYSGBL	Pages form a system global section. By default, pages form a group global section.

You must not set either the SEC$M_CRF (copy-on-reference) or the SEC$M_DZRO (demand-zero) bit when mapping by page frame number.

[pagcnt]
Number of pages in the section; the value of this argument must not be zero.

[vbn]
Page frame number of the first page to be mapped (as opposed to this argument's normal usage identifying the starting virtual block number (vbn) within a disk file). When you are mapping more than one page with a single $CRMPSC system service request, the pages are physically contiguous starting with the specified page.

19.2.1 Notes on PFN Mapping

The following considerations apply to PFN mapping.

1 An error in mapping UNIBUS or Q22 bus adapter address space or a reference to a nonexistent bus address causes a UNIBUS adapter error. However, this error does not cause a system failure (except on a VAX 85*x*0/8700/88*x*0, VAX–11/750, or VAX–11/730 system, where a machine check will occur). Rather, an entry is made in the system error log file and the user program continues executing (probably with erroneous results). The process is not notified of the UNIBUS adapter error.

2 On systems where a UNIBUS power failure can occur without causing a system failure, a user process receives a machine check exception, if it is using process space mapping when accessing UNIBUS or Q22 bus I/O address space during the failure. To survive this exception, the process must have a condition handler to deal with machine check exceptions. The *VMS System Services Reference Manual* discusses condition handlers in detail.

3 During recovery from a power failure, the processor spends a considerable amount of time (perhaps 10 to 60 milliseconds) at IPL 31. This action blocks user processes from executing during the recovery.

4 When a process requests deletion of a PFN-mapped page, VMS will wait until there is no direct I/O outstanding for the process before deleting the page. This is because no reference count is maintained for PFN-mapped pages. (For example, VMS cannot determine whether outstanding direct I/O is for the PFN-mapped page or not.) Applications using devices that have direct I/O perpetually outstanding, such as the DR32, must not delete PFN-mapped pages because this will cause the process to hang in the MWAIT state.

Once you have mapped to I/O address space, you can read data from a device data buffer register, because the device registers are now addressable as part of your process's virtual memory. The UNIBUS adapter performs the actual mapping of VAX virtual addresses to the 18-bit UNIBUS addresses that correspond to device registers. Likewise, the MicroVAX systems perform the mapping of virtual addresses to 22-bit Q22 bus addresses that correspond to device registers.

See Section 5.2 for a list of restrictions that apply to instruction references to device register address space.

19.3 Connecting to an Interrupt Vector

You can use the $QIO system service with an appropriate function code to connect to a device interrupt vector and to specify a user-supplied interrupt service routine that VMS executes when the designated device interrupts. Connecting to a device interrupt vector allows you to do the following:

- Respond to an interrupt within a short time

- Preempt other system processing to handle a real-time event, for example, a clock interrupt
- Buffer data from a device in real time and return the data to the process at a later time
- Set an event flag or queue an AST to your process after receiving the interrupt

An interrupt service routine specified in your process allows it to perform some of the functions normally performed by a device driver. The connect-to-interrupt facility, with its VMS-supplied driver (CONINTERR), thus allows you to avoid writing a full device driver and loading it into the operating system.

If you must access device registers from user mode (that is, from the main program or a user-mode AST procedure), you must use the Create and Map Section ($CRMPSC) system service to map I/O address space, specifying page frame number (PFN) mapping. The service creates a global or private section that maps the specified I/O pages into your process's virtual address space with suitable protection. The process can then gain access to I/O address space using perprocess virtual addresses (see Section 19.1 for additional discussion).

You do not need to map I/O address space to access device registers from any of the following routines specified in the $QIO call connecting to an interrupt vector:

- Unit initialization routine
- Start-I/O routine
- Interrupt service routine
- Cancel-I/O routine

These routines execute in system context and thus can access UNIBUS or Q22 bus I/O address space, which is mapped as part of system address space.

19.3.1 Performing the Connect-to-Interrupt

Connecting to a device interrupt vector allows your program to receive notification of an interrupt from a designated device by any combination of the following means:

- By execution of a user-supplied interrupt service routine
- By the setting of an event flag
- By execution of an AST procedure that gains control in process context

In addition, you can specify a cancel-I/O routine that is executed when the process disconnects from the interrupt vector or is deleted.

Before your program can run, the system manager must have performed the following actions at system generation time:

- Specify the *REALTIME_SPTS* system parameter, reserving system page table entries for use by real-time processes. These system page-table entries are used to map process-specified buffers in system address space (see the **p1** argument description in Section 19.3.2). The *REALTIME_SPTS* parameter value must be greater than or equal to the number of pages in buffers specified by processes connected to interrupt vectors.
- Configure the real-time device by issuing a CONNECT command to the System Generation Utility. This command names the device; its vector, register, and adapter addresses; and a skeletal driver (CONINTERR) for the device. (See the description of the CONNECT command in Section 12.2.2 and in the *VMS System Generation Utility Manual.*)

At run time the process calls the $ASSIGN system service to associate a channel with the device. To connect to the device interrupt vector, the process issues a $QIO call specifying the IO$_CONINTREAD or IO$_CONINTWRITE function code and as many of the following items as are appropriate:

- An interrupt service routine to be executed when the device generates an interrupt.
- A unit initialization routine.
- A start-I/O routine.
- A cancel-I/O routine.
- A buffer containing the code to be executed in system context, data (that is, the previously-listed routines), or both.
- An AST procedure to execute, an event flag to be set after the interrupt service routine (if any) completes, or both. (If an AST procedure is specified, an AST parameter may also be specified.)

A nonprivileged process (that is, lacking the CMKRNL privilege) can also connect to an interrupt vector, but it can only specify an AST procedure to be executed or an event flag to be set (or both) when an interrupt is generated. The process can also map the page in UNIBUS or Q22 bus I/O address space containing the device registers (see Section 19.2).

19.3.2 $QIO Connect-to-Interrupt Request to Driver

The format of the $QIO system service to connect to an interrupt vector follows. This explanation is limited to connecting to an interrupt vector. For a detailed description of the $QIO system service, see the *VMS System Services Reference Manual.*

VAX MACRO Format

```
$QIO   [efn] ,[chan] ,func [,iosb] [,astadr] [,astprm] -
       [,p1] [,p2] [,p3] [,p4] [,p5] [,p6]
```

High-Level Language Format

SYS$QIO ([efn] ,[chan] ,func [,iosb] [,astadr] [,astprm]
[,p1] [,p2] [,p3] [,p4] [,p5] [,p6])

Arguments

[efn]
[chan]
[iosb]
[astadr]
[astprm]

These arguments apply to the $QIO system service completion, not to device interrupt actions. For an explanation of these arguments, see the description of the $QIO system service in the *VMS System Services Reference Manual*.

func

Function code of IO$_CONINTREAD or IO$_CONINTWRITE. The IO$_CONINTWRITE function code allows locations in the buffer pointed to by the **p1** argument to be modified; the IO$_CONINTREAD function code makes the buffer contents read-only.

[p1]

Address of a descriptor for the buffer containing code and/or data. The first longword records the number of bytes in the buffer; the second longword records the address of the buffer. The buffer size must not exceed 65,535 bytes.

[p2]

Address of an entry point list. The list consists of four longwords that contain offsets into the buffer (specified in the **p1** argument) of the entry points of process-specified routines. These longwords and their contents[1] are as follows:

Symbol	Meaning
CIN$L_INIDEV	Offset to unit initialization routine
CIN$L_START	Offset to start-I/O routine
CIN$L_ISR	Offset to interrupt service routine
CIN$L_CANCEL	Offset to cancel-I/O routine

[p3]

Longword containing flags and an optional event flag number specification.

The low-order word contains the inclusive-OR of flags describing options to the connect-to-interrupt facility. The flags and their meanings are as follows:

[1] The listed symbols are defined by the $CINDEF macro located in the library SYS$LIBRARY:LIB.MLB.

Flag	Meaning
CIN$M_EFN	Set event flag on interrupt.
CIN$M_USECAL	Use CALL interface to process-specified routines (default is JSB interface).
CIN$M_REPEAT	Leave process connected to the interrupt vector until the connection is canceled.
CIN$M_INIDEV	Process-specified unit initialization routine is in the buffer specified in the **p1** argument.
CIN$M_START	Process-specified start-I/O routine is in buffer.
CIN$M_ISR	Process-specified interrupt service routine is in buffer.
CIN$M_CANCEL	Process-specified cancel-I/O routine is in buffer.

The high-order word specifies the number of the event flag to be set when an interrupt occurs. This number is expressed as an offset to CIN$V_EFNUM.

For example, to specify that your interrupt service routine is in the buffer and to set event flag 4, code **p3** as follows:

```
P3 = <CIN$M_ISR!CIN$M_EFN!4@CIN$V_EFNUM>
```

See note 3 in the following description for additional information on these flags.

[p4]
Address of the entry mask of an AST procedure to be called as the result of an interrupt (see Section 19.3.5).

[p5]
AST parameter to be passed to the AST procedure (used as the AST parameter only if the process-supplied interrupt service routine does not overwrite the value).

[p6]
Number of AST control blocks to preallocate in anticipation of fast, recurrent interrupts from the device.

Condition Values Returned

SS$_NORMAL	System service successfully completed.
SS$_ACCVIO	The caller does not have the appropriate access to the buffer specified in the **p1** argument or to the entry point list specified in the **p2** argument.
SS$_BADPARAM	The size of the buffer specified in the **p1** argument exceeds 65,535 bytes, or the number of preallocated AST control blocks specified in the P6 argument exceeds 65,535.
SS$_DISCONNECT	A connection is already outstanding for the device, or a condition described as follows in note 2b has occurred.
SS$_EXQUOTA	The process has exceeded its direct I/O limit quota or its AST limit quota.

SS$_ILLEFC	An illegal event flag number was specified.
SS$_INSFMEM	Insufficient system dynamic memory is available to complete the system service.
SS$_INSFSPTS	Insufficient system page-table entries are available to double map the process buffer. (The value of the *REALTIME_SPTS* SYSGEN parameter must be increased.)
SS$_NOPRIV	The process does not have the CMKRNL privilege. This privilege is only required if the user specifies a buffer to be used by the process and the process-specified kernel-mode routines.
SS$_UNASEFC	The process is not associated with the cluster containing the specified event flag.

Privilege Restrictions

The connect-to-interrupt $QIO call does not require privileges if no shared buffer is specified. If the request specifies a buffer descriptor argument (that is, **p1**), the process must have the CMKRNL privilege.

Resources Required/Returned

A connect-to-interrupt request updates the process quota values as follows:

- Subtracts the number of preallocated AST control blocks in the **p6** argument from the number of outstanding ASTs remaining for the process (ASTCNT)
- Subtracts 1 (for the $QIO) from the direct I/O count (DIOCNT)

Notes

1 After the $QIO call is issued, the operation is not completed until the process or the connect-to-interrupt driver cancels I/O on the channel.

2 The connect-to-interrupt driver can cancel I/O on the channel for a number of reasons, including the following:

   a. The driver cannot set the specified event flag, perhaps because the process disassociated from the common event flag cluster after requesting that a flag in that cluster be set.

   b. The driver cannot reallocate AST control blocks quickly enough. This condition can occur because not enough AST control blocks (**p6** argument) were specified, not enough pool space is available for the requested AST control blocks, or the process ASTCNT quota is exhausted.

   c. The driver cannot queue the AST to the process.

3 If no event flag setting was requested in the **p3** argument and if no AST procedure was specified in the **p4** argument, **p6** is ignored and no AST control blocks are preallocated. If you requested that an event flag be set or specified an AST procedure, but did not preallocate any AST control blocks (that is, **p6** is zero), one AST control block is preallocated automatically, because the system needs one control block to set any event flag or to deliver any ASTs.

If you request an event flag and/or an AST procedure and if you preallocate any AST control blocks, the CIN$M_REPEAT bit is set automatically in the longword specified in the **p3** argument. Thus, as long as you preallocate any AST control blocks, your process will automatically remain connected to the interrupt vector to receive repeated interrupts until the process is disconnected from the interrupt vector.

If the CIN$M_REPEAT flag is not set, the process is disconnected from the interrupt vector after the first successful interrupt, and a status code of SS$_NORMAL is returned.

19.3.3 The Connect-to-Interrupt Driver (CONINTERR.EXE)

The VMS connect-to-interrupt driver (CONINTERR) provides a driver interface to the system on behalf of the process. CONINTERR connects the process to the device by executing the following steps:

1. Validates the arguments to the $QIO system service call, such as the accessibility of the buffer specified in argument **p1** to the process, and the number of the event flag optionally specified in the **efn** argument.
2. Locks the physical pages of the buffer into physical memory, and maps the pages using system page-table entries allocated by the *REALTIME_SPTS* SYSGEN parameter.
3. Constructs argument lists and calling interfaces to the process-specified routines by storing values in the device's unit control block (UCB).
4. Allocates the specified number of AST control blocks to the process, and inserts each block in a queue in the device's UCB.
5. Transfers control to VMS to queue the connect-to-interrupt I/O packet to the CONINTERR start-I/O routine.

When the CONINTERR start-I/O routine gains control, it passes control, by means of a user-specified JSB or CALLS instruction interface, to the process-specified start-I/O routine. This routine usually initializes the device and may also start device activity.

When the device generates an interrupt, the CONINTERR interrupt service routine gains control. This routine transfers control to the process-supplied interrupt service routine.

19.3.4 Process-Specified Routines

Any routines that the process specifies in the connect-to-interrupt call, with the exception of the AST procedure, are double-mapped, once in process address space and once in system address space. Each routine executes in kernel mode at an appropriate IPL, as listed in the following table.

Routine	IPL
Unit initialization routine (after power recovery)	IPL$_POWER (IPL 31)
Start-I/O routine	IPL$_QUEUEAST (IPL 6)
Interrupt service routine	Device IPL
Cancel-I/O routine	IPL$_QUEUEAST (IPL 6)

The process must have CMKRNL privilege. Each routine must

- Be position independent
- Follow the rules for accessing I/O address space as described in Section 5.2
- Access only data within the buffer or nonpageable locations in system address space
- Perform any necessary synchronization of access to data in the shared buffer
- Save any registers it uses (unless otherwise noted in the remaining sections of this chapter)
- Exit properly
- Not incur exceptions
- Not perform lengthy processing
- Not dispatch to code outside the buffer specified in the **p1** argument to the $QIO system service call

Only VAX MACRO or VAX BLISS-32 should be used to code process-specified routines in system address space or any references to I/O address space. There is no assurance that the code generated by compilers for other languages will satisfy all the constraints described in this section.

The following constraints apply to process-specified routines in system address space (that is, in the buffer specified in the **p1** argument to the $QIO call that establishes the connection to the interrupt vector):

- The compiler must generate position-independent code for the routines.
- The generated code and data must be contiguous in virtual address space.
- No calls can be made to any procedure outside the buffer. (This restriction includes calls to routines in the VAX Run-Time Library.)
- For any references to I/O address space, the generated code must follow the rules for accessing I/O address space discussed in Section 5.2.

You can find additional help for writing a start-I/O routine, interrupt service routine, unit initialization routine, or cancel-I/O routine in Sections 8, 9, 11.1, and 11.2, respectively. Additionally, you may find useful the several program examples of connecting to an interrupt vector with which this chapter concludes.

19.3.4.1 Unit Initialization Routine

During recovery from a power failure, VMS calls the CONINTERR unit initialization routine. This routine marks the device as on line in the UCB$L_STS field, stores the UCB address in the IDB$L_OWNER field, and then transfers control to the process-specified unit initialization routine. The process-specified routine executes in system context at IPL$_POWER (IPL 31). If the process specified a JSB interface, the process unit initialization routine gains control with the following register settings:

R0	Address of UCB
R4	Address of CSR
R5	Address of IDB
R6	Address of DDB
R8	Address of CRB

If the process specified a CALL interface, the process unit initialization routine gains control with an argument list pointed to by AP:

00(AP)	Argument count of 5
04(AP)	Address of CSR
08(AP)	Address of IDB
12(AP)	Address of DDB
16(AP)	Address of CRB
20(AP)	Address of UCB

The process-specified unit initialization routine may initialize device registers. It must follow these conventions:

- Not lower IPL nor obtain any spin locks
- Save and restore all registers it uses, other than R0 through R3
- Restore the stack to its original state before exiting
- Exit with an RSB instruction (for a JSB interface) or a RET instruction (for a CALL interface)

For more on writing a unit initialization routine, see Section 11.1.

19.3.4.2 Start-I/O Routine

The process-specified start-I/O routine executes in process context in system space at IPL$_QUEUEAST (IPL 6), holding the QUEUEAST fork lock in a VMS multiprocessing environment. It is entered from the CONINTERR start-I/O routine.

If the process specified a JSB interface, the process start-I/O routine gains control with the following register settings:

R2	Address of counted argument list
R3	Address of IRP
R5	Address of UCB

If the process specified a CALL interface, the process start-I/O routine gains control with an argument list pointed to by AP:

00(AP)	Argument count of 4
04(AP)	System-mapped address of process buffer
08(AP)	Address of IRP
12(AP)	System-mapped address of the device's CSR
16(AP)	Address of UCB

The process-specified start-I/O routine may set up device registers. It must follow these conventions:

- Maintain an IPL equal to or higher than IPL$_QUEUEAST (IPL 6), and exit at IPL 6. (If it raises IPL, the routine should first save the current IPL on the stack for later use in restoring IPL.) In a VMS multiprocessing system, the process-specified start-I/O routine must suitably synchronize any access of device registers with the process-specified interrupt service routine. To do so, each routine must obtain the appropriate device lock, using the VMS-supplied macro DEVICELOCK. Before exiting, each routine releases ownership of the device lock using the DEVICEUNLOCK macro. (See the discussion of these macros in the *VMS Device Support Reference Manual*.)
- Save and restore all registers it uses, other than R0 through R4.
- Restore the stack to its original state before exiting.
- Exit with an RSB instruction (for a JSB interface) or a RET instruction (for a CALL interface).

For additional information on writing a start-I/O routine, see Chapter 8.

19.3.4.3 Interrupt Service Routine

A process-specified interrupt service routine is entered when an interrupt from the device occurs. This routine executes in system context at device IPL.

If the process specified a JSB interface, the process interrupt service routine gains control with the following register settings:

R2	Address of counted argument list
R4	Address of IDB
R5	Address of UCB

If the process specified a CALL interface, the process interrupt service routine gains control with an argument list pointed to by AP:

00(AP)	Argument count of 5
04(AP)	System-mapped address of process buffer
08(AP)	Address of AST parameter
12(AP)	System-mapped address of the device's CSR
16(AP)	Address of IDB
20(AP)	Address of UCB

The process-specified interrupt service routine usually performs one or more of the following steps:

1 Copies the contents of device registers into the shared buffer or the AST parameter

2 Writes to a device register to clear the interrupt condition, if such an operation is required for the device

3 Restarts the device, or returns an offset, a byte count, or actual data as an AST parameter

4 Returns an interrupt status to the VMS connect-to-interrupt driver (CONINTERR)

The process-specified interrupt service routine, like those supplied by VMS, has the following characteristics:

- It is mapped in system address space.
- It executes on the interrupt stack.
- It executes at the IPL of the device that requested the interrupt.

The routine must follow these conventions:

- Maintain an IPL equal to or higher than device IPL. (If it raises IPL, the routine should first save the current IPL on the stack for later use in restoring IPL.) In a VMS multiprocessing system, if the process-specified start-I/O routine or cancel-I/O routine accesses device registers or UCB fields also accessed by the process-specified interrupt service routine, the routines must suitably synchronize. To do so, each routine must obtain the appropriate device lock, using the VMS-supplied macro DEVICELOCK. Before exiting, each routine releases ownership of the device lock using the DEVICEUNLOCK macro. (See the discussion of these macros in the *VMS Device Support Reference Manual*.)
- Save and restore all registers it uses, other than R0 through R4.
- Restore the stack to its original state before exiting.
- Set or clear the low bit of R0, as a status value, before exiting. The status values are as follows:

Bit 0 of R0	Meaning
Clear	Dismiss the interrupt. The process is not notified of the interrupt.
Set	Set the event flag if CIN$M_EFN bit is set in the **p3** argument to the $QIO system service call, and queue the AST if **p4** specifies an AST procedure.

- Return to the CONINTERR interrupt service routine with a RET instruction (for a CALL interface) or RSB instruction (for a JSB interface).

Depending on the interrupt status returned in R0, the CONINTERR interrupt service routine queues a fork process to run at a lower IPL (IPL$_QUEUEAST). Then the interrupt service routine exits from the interrupt with an REI instruction. When the CONINTERR fork process gains control, it queues an AST or posts an event flag to the process (or both).

For additional information on writing an interrupt service routine, see Chapter 9.

19.3.4.4 Cancel-I/O Routine

When the user process issues a cancel-I/O request for a device connected to the process, the CONINTERR cancel-I/O routine first checks to determine whether the process can indeed cancel I/O for this device. If it can, the CONINTERR cancel-I/O routine transfers control to the process-specified cancel-I/O routine. This routine executes in system context at IPL 8 (fork IPL).

If the process specified a JSB interface, the process cancel-I/O routine gains control with the following register settings:

R2	Negated value of channel index number
R3	Address of current IRP
R4	Address of PCB for process canceling the I/O
R5	Address of UCB

If the process specified a CALL interface, the process cancel-I/O routine gains control with an argument list pointed to by AP:

00(AP)	Argument list count of 4
04(AP)	Negated value of channel index number
08(AP)	Address of current IRP
12(AP)	Address of PCB for process canceling the I/O
16(AP)	Address of UCB

The process-specified cancel-I/O routine may clear device registers and set the UCB$V_CANCEL bit in UCB$L_STS. It must follow these conventions:

- Maintain an IPL equal to or higher than IPL$_QUEUEAST (IPL 6), and exit at IPL 6. (If it raises IPL, the routine should first save the current IPL on the stack for later use in restoring IPL.) In a VMS multiprocessing system, if the process-specified cancel-I/O routine accesses device registers or UCB fields also accessed by the process-specified interrupt service routine, the routines must suitably synchronize. To do so, each routine must obtain the appropriate device lock, using the VMS-supplied macro DEVICELOCK. Before exiting, each routine releases ownership of the device lock using the DEVICEUNLOCK macro. (See the discussion of these macros in the *VMS Device Support Reference Manual*.)
- Save and restore all registers it uses, other than R0 through R3.

- Place a completion status in R0 and R1. VMS places the values in these registers in the I/O status block associated with the connect-to-interrupt $QIO call.
- Restore the stack to its original state before exiting.
- Exit with an RSB instruction (for a JSB interface) or a RET instruction (for a CALL interface).

For additional information on writing a cancel-I/O routine, see Section 11.2.

19.3.5 AST Procedure

The AST procedure that you specify in the call to the $QIO system service for the connect-to-interrupt operation gains control in process context. This routine usually performs one or more of the following steps:

1 Reads or writes device registers if the process mapped I/O address space.

2 Interprets data. Use caution, however, because any processing done by the AST procedure can be interrupted by a device interrupt, which might store more data or modify the buffer's contents.

3 Calls the Cancel I/O on Channel ($CANCEL) system service to disconnect the process from the interrupt. Once the process is completely disconnected, the CONINTERR driver clears all interrupts for the driver.

19.4 Real-Time Applications Examples

To understand how the connect-to-interrupt facility is useful for programming real-time devices, consider devices used in three types of real-time applications:

1 Asynchronous event reporting without data—devices that generate an interrupt as the result of an external event not initiated by a programmed request.

2 Program-driven data collection—devices that generate an interrupt as the result of a programmed request, and make the result of the request available as data in a device register at the time of the interrupt.

3 Asynchronous event reporting with data—one device triggers another device by generating an interrupt that causes a programmed request to be sent to the other device, which in turn generates an interrupt.

Examples of these three types of real-time applications and models of programs to handle the devices follow.

Note: The configurations described in the examples in this section are not officially supported; Digital does not provide device driver, UETP, or diagnostic support for certain devices mentioned. (In fact, Digital has officially retired the -K series models (AD11-K and AM11-K A/D Converter). The examples are provided

merely as possible models for users who wish to design real-time applications using unsupported devices or configurations.

The files in the SYS$EXAMPLES directory whose names begin with "LABIO" illustrate an application using the connect-to-interrupt technique. Included is a program example illustrating data definitions and coding used to connect to a device interrupt vector.

19.4.1 Example 1: KW11–W Watchdog Timer

This type of device reports asynchronous external events: it generates an interrupt as a result of an external event not initiated by a programmed request. The only data of interest to be passed to the user process is the occurrence of the external event. Such devices include contact and/or solid state interrupts, and clocks or counters. The program may need to activate clock and counter devices by means of a programmed request, but any subsequent interrupts are the result of external events only.

In this example, a dual-processor system uses two KW11–W watchdog timers connected back-to-back to monitor CPU failures. Each processor must arm its timer at regular intervals to prevent the timer from operating a relay that outputs an alarm signal. The alarm output of each timer is connected to the receive input of the other watchdog. If processor A fails and its watchdog times out, the alarm output generates an interrupt on processor B by way of the second watchdog timer.

The watchdog control program on each processor simply addresses the timer at regular intervals. If the interval passes without the timer being addressed, the timer operates an output relay that generates an interrupt to the second CPU. For this example, assume that the interval is 5 seconds. (Section 19.4.3 contains an example that addresses the problem of a much smaller time interval.)

The watchdog control program on processor A executes as follows:

1. Assigns a channel to the device
2. Calls the $CRMPSC system service to map to the I/O page in order to address the device registers
3. Issues a connect-to-interrupt $QIO request to connect the program to the watchdog timer for processor B; specifies the addresses of an interrupt service routine and an AST procedure
4. Writes a value to a device register to start the timer
5. Calls the $SETIMR system service to request that an event flag be set after a specified interval (for example, 4 seconds)
6. Calls the $WAITFR system service to wait for the event flag
7. When the event flag is set, writes a value to a device register to reset the timer
8. Loops to step 5

The same control program runs on processor B except that it connects to the watchdog timer for processor A. If either processor fails, the watchdog timer generates an interrupt on the other processor.

The standby processor that receives the interrupt gains control in the VMS connect-to-interrupt driver (CONINTERR), which calls a process-supplied interrupt service routine (defined in step 3) that handles the interrupt as follows:

1 Sets the KW11–W switch relay register to clear the timer interrupt condition

2 Sets a status flag that will cause an AST to be delivered to the control program that connected to the interrupt

3 Returns to CONINTERR

CONINTERR completes the interrupt handling as follows:

1 Schedules a fork process at a lower IPL (IPL$_QUEUEAST). This fork process, when it gains control, will queue an AST to the user program.

2 Executes an REI instruction to return from the interrupt.

The timer control program on the standby processor regains control in an AST procedure which responds to the other processor's failure by switching over and assuming control of the other processor's tasks (or whatever is appropriate).

19.4.2 Example 2: AD11–K, AM11–K A/D Converter with Multiplexer Connected to the UNIBUS

This type of device provides program-driven data collection: it generates an interrupt as the result of a programmed request to the device, and makes the result of the request available as data in a device register. Typical devices include A/D converters and Digital I/O registers.

The data collection operation is usually repetitive for such applications. Therefore, the interrupt service routine must be capable of buffering data from the device in order to ensure that no data is lost because of the high-speed data transfer rate. A typical buffer size for this sampling technique might be 32 16-bit words.

In this example, a user program controls an AD11–K/AM11–K combination that accepts analog data from thermocouples. The AD11–K converts analog data to Digital data and returns the data in a device register. Every 10 seconds, the program samples 16 to 32 out of 64 channels at gain settings that may vary based on the thermocouple type and previous samplings.

To collect data efficiently, the program buffers data in a process-specified interrupt service routine, and requests delivery of an AST to the user process when all the requested channels have been sampled. To perform variable sampling, the program passes parameters to the interrupt service routine.

The program establishes a protocol to communicate between the program and the interrupt service routine. The protocol defines a data area shared by the main program, the interrupt service routine, and the AST procedure. The data area contains parameters from the program and data from the AD11–K. The data area is a 98-word array used as follows:

1 Elements 1 and 2 of the data area contain an index to the next buffer location to be filled, and a count indicating the number of samplings still to be taken. The main program initializes these values before starting the device. The interrupt service routine reads and modifies these values in the process of copying data and determining when to stop sampling.

2 Elements 3 to 66 of the data area are reserved for interrupt service routine parameters. Each pair of elements contains the number of a channel and a gain value. The main program loads these parameters before starting the device.

3 Elements 67 to 98 of the data area receive the data that the interrupt service routine reads from the AD11–K data buffer register. The AST routine later reads data from this part of the buffer.

The program sets up for the sampling as follows:

1 Assigns a channel to the device

2 Calls the $CRMPSC system service to map to the I/O page in order to address the device registers

3 Initializes the data area by writing a 67 (the index to the next buffer location to be filled) into element 1, and the number of samples to take into element 2 of the data area; clears elements 3 through 98 of the data area

4 Writes channel numbers and gain values into the parameter section of the data area

5 Issues a connect-to-interrupt $QIO call to connect the process to the A/D converter; specifies the addresses of the area to be double-mapped, an offset to the interrupt service routine, and an AST procedure

6 Sets the start and interrupt-enable bits in the AD11–K status register to start the A/D converter

7 Calls the $HIBER system service to place the process in a wait state

As soon as the AD11–K has converted the first sample, the device generates an interrupt. CONINTERR.EXE calls the process-specified interrupt service routine. This process-specified routine executes as follows:

1 Computes the next location to be written in the buffer by reading the first element in the data area

2 Reads 12 bits of data from the A/D buffer register into the next location in the buffer

3 Updates the buffer offset and count elements at the beginning of the data area

4 If all requested samples have been collected, writes the address of the data area into the AST parameter, sets a status flag that will cause an AST to be delivered to the control program, and returns to the CONINTERR routine

5 Otherwise, sets the start bit in a device register to restart the device and returns to the CONINTERR routine with a status flag requesting no AST delivery or event flag setting

Based on the interrupt status from the process-specified interrupt service routine, the CONINTERR routine completes the interrupt processing by queuing a fork process that will queue an AST to the user process. When the process gains control in the AST procedure, it processes the samples in the following steps:

1 Clears the interrupt-enable bit in the device status register

2 Examines the data collected in order to adjust channel selection and/or gain values for the next sampling

3 Copies the data to a file

4 Reinitializes the data area

5 Calls the $SCHDWK system service to wake the process after a short interval (for example, 10 seconds)

6 Returns

When the time interval elapses, the process regains control. The program can then restart the sampling process by again setting the start and interrupt-enable bits in the AD11–K status register.

19.4.3 Example 3: KW11–P Real-Time Clock and AD11–K Converter Connected to the UNIBUS

This type of device reports asynchronous external events by collecting data: one device triggers another device by generating an interrupt that causes a programmed request to be sent to the other device, which in turn generates an interrupt. A typical example is a clock-driven A/D operation for precise time sampling as required in signal processing. This processing technique is often used in laboratories. The amount of data collected in such a timed sampling might typically be 200 to 1000 16-bit words.

In this example, the main program sets up the real-time clock to generate interrupts periodically. At regular intervals, the clock interrupt triggers a programmed request for an A/D conversion operation. The AD11–K collects a sample, and interrupts the CPU with a "done" interrupt and 12 bits of data. The AD11–K interrupt service routine buffers the data and, if the buffer is full, causes an AST to be delivered to the process. The process, gaining control in an AST procedure, copies the buffered data to another buffer or to disk.

Programming these device functions is slightly more complicated than the previous example. The main program must specify a large buffer to be used in ring fashion to guarantee that data is not lost between clock-driven samplings. In addition, the program must connect to two device interrupts—one for the clock and one for the A/D converter.

The protocol used by the main program, the interrupt service routine, and the AST procedure is similar to the previous example. The data area is larger: 4K words of buffer area follow the parameter area. The A/D converter interrupt service routine and the AST procedure treat the 4K-word buffer as four buffer sections of 1K words per section. The first element in each 1K buffer section is a flag indicating whether the section is in use. The AST resets the flag value after copying the contents of the buffer. The interrupt service routine uses a buffer section only if the section's flag value indicates that the buffer has been emptied.

The main program starts the sampling with the following steps:

1 Assigns channels to the clock and to the A/D converter.

2 Calls the $CRMPSC system service to map to the I/O page in order to address the device registers.

3 Initializes the data buffer by writing a 67 (the index to the next buffer location to be filled) into element 1, and the number of samples to take into element 2 of the data area; clears elements 3 through 4096 of the data area; flags each page of the buffer as available.

4 Writes channel numbers and gain values into the parameter segments of the data area.

5 Issues a connect-to-interrupt $QIO call to connect the process to the clock, and specifies the address of an interrupt service routine.

6 Issues a connect-to-interrupt $QIO call to connect the process to the A/D converter; and specifies the addresses of the area to be double mapped, an offset to the interrupt service routine and an AST procedure.

7 Sets the sampling interval by writing a 16-bit value into the KW11–P count set buffer register.

8 Starts the clock by setting the run, mode, rate selection, and interrupt-enable bits in the KW11–P control and status register. Setting the mode bit causes repeated interrupts generated at a rate specified in the time interval.

9 Calls the $HIBER system service to place the process in a wait state.

The clock interrupts when zero (underflow) occurs during a countdown from the preset interval count. The VMS CONINTERR routine calls the process-specified clock interrupt service routine. This process-specified routine starts the A/D conversion as follows:

1 Starts the A/D converter by setting the start and interrupt-enable bits in the AD11–K status register

2 Sets interrupt status that prevents AST delivery or event flag setting as a result of this interrupt

3 Returns to CONINTERR

Starting the A/D converter results in an interrupt from the AD11–K, and control passes, by way of CONINTERR, to the AD11–K interrupt service routine. This routine executes as follows:

1 If this sample is the first sample for a new buffer (indicated by a flag in the data area), the routine moves to the next buffer section (branching to error handling if the buffer is still full), and sets up the first two elements of the data area to indicate the buffer section to be written next. Then it sets the flag at the start of the new buffer section and sets a flag in the data area to indicate that sampling is occurring.

2 The routine computes the next location to be written in the buffer by reading the first location in the data area.

3 The routine reads 12 bits of data from the A/D buffer register into the next location in the buffer.

4 The routine updates the buffer offset and count values in the data area.

5 If this sample fills the data sector, the routine writes the offset of the filled sector from the start of the 4K-word buffer into the AST parameter, sets a status flag that will cause an AST to be delivered to the control program, and sets a flag indicating that a new data section is to be started.

6 The routine returns to CONINTERR.

The AST procedure copies and fills the next buffer section with zeros to indicate that the section is again available to the interrupt service routine. When the next clock interrupt occurs, the data can be written to the next buffer section, even if the AST routine has not yet emptied the previous buffer section.

A VMS Driver Template

This appendix lists the contents of the VMS driver template. The code in this template can serve as the starting point for a new UNIBUS or Q22 bus device driver. You can obtain a machine-readable copy of this driver from SYS$EXAMPLES:TDRIVER.MAR.

```
        .TITLE  TDRIVER - VAX/VMS TEMPLATE DRIVER
        .IDENT  'X-3'

;
;*****************************************************************************
;*                                                                           *
;*  COPYRIGHT (c) 1978, 1980, 1982, 1984 BY                                  *
;*  DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASSACHUSETTS.                   *
;*  ALL RIGHTS RESERVED.                                                     *
;*                                                                           *
;*  THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED AND COPIED    *
;*  ONLY IN  ACCORDANCE WITH  THE  TERMS  OF  SUCH  LICENSE  AND WITH THE    *
;*  INCLUSION OF THE ABOVE COPYRIGHT NOTICE. THIS SOFTWARE OR  ANY  OTHER    *
;*  COPIES THEREOF MAY NOT BE PROVIDED OR OTHERWISE MADE AVAILABLE TO ANY    *
;*  OTHER PERSON.  NO TITLE TO AND OWNERSHIP OF  THE  SOFTWARE IS  HEREBY    *
;*  TRANSFERRED.                                                             *
;*                                                                           *
;*  THE INFORMATION IN THIS SOFTWARE IS  SUBJECT TO CHANGE WITHOUT NOTICE    *
;*  AND  SHOULD  NOT  BE  CONSTRUED AS  A COMMITMENT BY DIGITAL EQUIPMENT    *
;*  CORPORATION.                                                             *
;*                                                                           *
;*  DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE  OR  RELIABILITY OF ITS    *
;*  SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DIGITAL.                  *
;*                                                                           *
;*                                                                           *
;*****************************************************************************
;

;++
;
; FACILITY:
;
;       VAX/VMS Template Driver
;
; ABSTRACT:
;
;       This module contains the outline of a driver:
;
;               Models of driver tables
;               Controller and unit initialization routines
;               An FDT routine
;               The start I/O routine
;               The interrupt service routine
;               The cancel I/O routine
;               The device register dump routine
;
; AUTHOR:
;
;       S. Programmer   11-Nov-1979
;
```

```
; REVISION HISTORY:
;
;       X-3     JHP003  J. Programmer    5-Oct-1988
;               Changed all user symbols to use underscores instead of
;               dollar signs to avoid conflict with any symbols defined
;               by Digital.  Added return status to unit initialization
;               routine.
;
;       X-2     JHP002  J. Programmer   21-Aug-1987
;               Add SMP support.
;
;       V02     JHP001  J. Programmer    2-Aug-1979      11:27
;               Remove BLBC instruction from CANCEL routine.
;
;       V02-001 ROW0067 R. Programmer   11-Feb-1981      13:10
;               Add description of reason argument to CANCEL routine.
;               Correct references to channel index number.
;
;--
        .SBTTL  External and local symbol definitions

;
; External symbols
;
        $CANDEF                         ; Cancel reason codes
        $CRBDEF                         ; Channel request block
        $DCDEF                          ; Device classes and types
        $DDBDEF                         ; Device data block
        $DEVDEF                         ; Device characteristics
        $DYNDEF                         ; Dynamic data structure definitions
        $IDBDEF                         ; Interrupt data block
        $IODEF                          ; I/O function codes
        $IPLDEF                         ; Hardware IPL definitions
        $IRPDEF                         ; I/O request packet
        $SSDEF                          ; System status codes
        $UCBDEF                         ; Unit control block
        $VECDEF                         ; Interrupt vector block

;
; Local symbols
;

;
; Argument list (AP) offsets for device-dependent QIO parameters
;
P1      = 0                             ; First QIO parameter
P2      = 4                             ; Second QIO parameter
P3      = 8                             ; Third QIO parameter
P4      = 12                            ; Fourth QIO parameter
P5      = 16                            ; Fifth QIO parameter
P6      = 20                            ; Sixth QIO parameter

;
; Other constants
;
TD_DEF_BUFSIZ   = 1024                  ; Default buffer size
TD_TIMEOUT_SEC  = 10                    ; 10-second device timeout
TD_NUM_REGS     = 4                     ; Device has 4 registers

;
; Definitions that follow the standard UCB fields
;
```

```
        $DEFINI UCB                     ; Start of UCB definitions

        .=UCB$K_LENGTH                  ; Position at end of UCB
$DEF    UCB_W_TD_WORD                   ; A sample word
                        .BLKW   1
$DEF    UCB_W_TD_STATUS                 ; Device's CSR
                        .BLKW   1
$DEF    UCB_W_TD_WRDCNT                 ; Device's word count register
                        .BLKW   1
$DEF    UCB_W_TD_BUFADR                 ; Device's buffer address
                        .BLKW   1       ; register
$DEF    UCB_W_TD_DATBUF                 ; Device's data buffer register
                        .BLKW   1
$DEF    UCB_K_TD_UCBLEN                 ; Length of extended UCB

;
; Bit positions for device-dependent status field in UCB
;
        _VIELD  UCB,0,<-                ; Device status
                <BIT_ZERO,,M>,-         ; First bit
                <BIT_ONE,,M>,-          ; Second bit
                >

        $DEFEND UCB                     ; End of UCB definitions

;
; Device register offsets from CSR address
;
        $DEFINI TD                      ; Start of status definitions

$DEF    TD_STATUS                       ; Control/status
                        .BLKW   1

;
; Bit positions for device control/status register
;
        _VIELD  TD_STS,0,<-             ; Control/status register
                <GO,,M>,-               ; Start device
                <BIT1,,M>,-             ; Bit 1
                <BIT2,,M>,-             ; Bit 2
                <BIT3,,M>,-             ; Bit 3
                <XBA,2,M>,-             ; Extended address bits
                <INTEN,,M>,-            ; Enable interrupts
                <READY,,M>,-            ; Device ready for command
                <BIT8,,M>,-             ; Bit 8
                <BIT9,,M>,-             ; Bit 9
                <BIT10,,M>,-            ; Bit 10
                <BIT11,,M>,-            ; Bit 11
                <,1>,-                  ; Disregarded bit
                <ATTN,,M>,-             ; Attention bit
                <NEX,,M>,-              ; Nonexistent memory flag
                <ERROR,,M>,-            ; Error or external interrupt
        >
$DEF    TD_WRDCNT                       ; Word count
                        .BLKW   1
$DEF    TD_BUFADR                       ; Buffer address
                        .BLKW   1
$DEF    TD_DATBUF                       ; Data buffer
                        .BLKW   1

        $DEFEND TD                      ; End of device register
                                        ; definitions.
```

```
        .SBTTL  Standard tables

;
; Driver prologue table
;

        DPTAB   -                                       ; DPT-creation macro
                END=TD_END,-                            ; End of driver label
                ADAPTER=UBA,-                           ; Adapter type
                UCBSIZE=<UCB_K_TD_UCBLEN>,-             ; Length of UCB
                NAME=TDDRIVER                           ; Driver name
        DPT_STORE INIT                                  ; Start of load
                                                        ; initialization table
        DPT_STORE UCB,UCB$B_FLCK,B,SPL$C_IOLOCK8        ; Device FORK LOCK
        DPT_STORE UCB,UCB$B_DIPL,B,22                   ; Device interrupt IPL
        DPT_STORE UCB,UCB$L_DEVCHAR,L,<-                ; Device characteristics
                DEV$M_IDV!-                             ;   input device
                DEV$M_ODV>                              ;   output device
        DPT_STORE UCB,UCB$B_DEVCLASS,B,DC$_SCOM         ; Sample device class
        DPT_STORE UCB,UCB$W_DEVBUFSIZ,W,-               ; Default buffer size
                TD_DEF_BUFSIZ

        DPT_STORE REINIT                                ; Start of reload
                                                        ; initialization table
        DPT_STORE DDB,DDB$L_DDT,D,TD$DDT                ; Address of DDT
        DPT_STORE CRB,CRB$L_INTD+VEC$L_ISR,D,-          ; Address of interrupt
                TD_INTERRUPT                            ; service routine
        DPT_STORE CRB,-                                 ; Address of controller
                CRB$L_INTD+VEC$L_INITIAL,-              ; initialization routine
                D,TD_CONTROL_INIT
        DPT_STORE CRB,-                                 ; Address of device
                CRB$L_INTD+VEC$L_UNITINIT,-             ; unit initialization
                D,TD_UNIT_INIT                          ; routine

        DPT_STORE END                                   ; End of initialization
                                                        ; tables

;
; Driver dispatch table
;

        DDTAB   -                                      ; DDT-creation macro
                DEVNAM=TD,-                            ; Name of device
                START=TD_START,-                       ; Start I/O routine
                FUNCTB=TD_FUNCTABLE,-                  ; FDT address
                CANCEL=TD_CANCEL,-                     ; Cancel I/O routine
                REGDMP=TD_REG_DUMP                     ; Register dump routine

;
; Function decision table
;
```

```
TD_FUNCTABLE:                                   ; FDT for driver
        FUNCTAB ,-                              ; Valid I/O functions
                <READVBLK,-                     ; Read virtual
                READLBLK,-                      ; Read logical
                READPBLK,-                      ; Read physical
                WRITEVBLK,-                     ; Write virtual
                WRITELBLK,-                     ; Write logical
                WRITEPBLK,-                     ; Write physical
                SETMODE,-                       ; Set device mode
                SETCHAR>                        ; Set device characteristics
        FUNCTAB ,                               ; No buffered functions
        FUNCTAB +EXE$READ,-                     ; FDT read routine for
                <READVBLK,-                     ; read virtual,
                READLBLK,-                      ; read logical,
                READPBLK>                       ; and read physical block
        FUNCTAB +EXE$WRITE,-                    ; FDT write routine for
                <WRITEVBLK,-                    ; write virtual,
                WRITELBLK,-                     ; write logical,
                WRITEPBLK>                      ; and write physical block
        FUNCTAB +EXE$SETMODE,-                  ; FDT set mode routine
                <SETCHAR,-                      ; set characteristics, and
                SETMODE>                        ; set mode


        .SBTTL  TD_CONTROL_INIT, Controller initialization routine

;++
; TD_CONTROL_INIT, Readies controller for I/O operations
;
; Functional description:
;
;       The operating system calls this routine in 3 places:
;
;               at system startup
;               during driver loading and reloading
;               during recovery from a power failure
;
; Inputs:
;
;       R4      - address of the CSR (controller status register)
;       R5      - address of the IDB (interrupt data block)
;       R6      - address of the DDB (device data block)
;       R8      - address of the CRB (channel request block)
;
; Outputs:
;
;       The routine must preserve all registers except R0-R3.
;
;--

TD_CONTROL_INIT:                        ; Initialize controller
        RSB                             ; Return

        .SBTTL  TD_UNIT_INIT, Unit initialization routine
```

```
;++
; TD_UNIT_INIT, Readies unit for I/O operations
;
; Functional description:
;
;       The operating system calls this routine after calling the
;       controller initialization routine:
;
;               at system startup
;               during driver loading
;               during recovery from a power failure
;
; Inputs:
;
;       R4      - address of the CSR (controller status register)
;       R5      - address of the UCB (unit control block)
;
; Outputs:
;
;       The routine must preserve all registers except R0-R3.
;
;--
TD_UNIT_INIT:                           ; Initialize unit
        BISW    #UCB$M_ONLINE, -
                UCB$W_STS(R5)           ; Set unit online
        MOVL    # SS$_NORMAL, R0       ; VAXBI devices need to return success
                                        ; if unit is properly configured.
        RSB                             ; Return

        .SBTTL  TD_FDT_ROUTINE, Sample FDT routine

;++
; TD_FDT_ROUTINE, Sample FDT routine
;
; Functional description:
;
;       T.B.S.
;
; Inputs:
;
;       R0-R2   - scratch registers
;       R3      - address of the IRP (I/O request packet)
;       R4      - address of the PCB (process control block)
;       R5      - address of the UCB (unit control block)
;       R6      - address of the CCB (channel control block)
;       R7      - bit number of the I/O function code
;       R8      - address of the FDT table entry for this routine
;       R9-R11  - scratch registers
;       AP      - address of the 1st function dependent QIO parameter
;
; Outputs:
;
;       The routine must preserve all registers except R0-R2, and
;       R9-R11.
;
;--
TD_FDT_ROUTINE:                         ; Sample FDT routine
        RSB                             ; Return


        .SBTTL  TD_START, Start I/O routine
```

```
;++
; TD_START - Start a transmit, receive, or set mode operation
;
; Functional description:
;
;       T.B.S.
;
; Inputs:
;
;       R3      - address of the IRP (I/O request packet)
;       R5      - address of the UCB (unit control block)
;
; Outputs:
;
;       R0      - 1st longword of I/O status: contains status code and
;                 number of bytes transferred
;       R1      - 2nd longword of I/O status: device-dependent
;
;       The routine must preserve all registers except R0-R2 and R4.
;
;--
TD_START:                                       ; Process an I/O packet
        DEVICELOCK LOCKADDR=UCB$L_DLCK(R5),-    ; Lock device access.
                SAVIPL=-(SP)                    ; Save current IPL

        WFIKPCH TD_TIMEOUT,#TD_TIMEOUT_SEC
;
; After a transfer completes successfully, return the number of bytes
; transferred and a success status code.
;
        IOFORK
        INSV    UCB$W_BCNT(R5),#16,-    ; Load number of bytes trans-
                #16,R0                  ; ferred into high word of R0.
        MOVW    #SS$_NORMAL,R0          ; Load a success code into R0.

;
; Call I/O postprocessing.
;
COMPLETE_IO:                            ; Driver processing is finished.
        REQCOM                          ; Complete I/O.

;
; Device timeout handling. Return an error status code.
;
TD_TIMEOUT:                                     ; Timeout handling
        DEVICEUNLOCK LOCKADDR=UCB$L_DLCK(R5),-  ; Unlock device access
                NEWIPL=#8,-                     ; Lower IPL
                PRESERVE=NO                     ; Don't preserve R0
        MOVZWL  #SS$_TIMEOUT,R0                 ; Return error status.
        BSBB    COMPLETE_IO                    ; Call I/O postprocessing.
        DEVICELOCK LOCKADDR=UCB$L_DLCK(R5),-    ; Acquire device lock for exit
                PRESERVE=NO
        RSB

        .SBTTL  TD_INTERRUPT, Interrupt service routine
```

```
;++
; TD_INTERRUPT, Analyzes interrupts, processes solicited interrupts
;
; Functional description:
;
;       The sample code assumes either
;
;               that the driver is for a single-unit controller, and
;               that the unit initialization code has stored the
;               address of the UCB in the IDB; or
;
;               that the driver's start I/O routine acquired the
;               controller's channel with a REQPCHANL macro call, and
;               then invoked the WFIKPCH macro to keep the channel
;               while waiting for an interrupt.
;
; Inputs:
;
;        0(SP)  - pointer to the address of the IDB (interrupt data
;                 block)
;        4(SP)  - saved R0
;        8(SP)  - saved R1
;       12(SP)  - saved R2
;       16(SP)  - saved R3
;       20(SP)  - saved R4
;       24(SP)  - saved R5
;       28(SP)  - saved PC
;       32(SP)  - saved PSL (processor status longword)
;
;       The IDB contains the CSR address and the UCB address.
;
; Outputs:
;
;       The routine must preserve all registers except R0-R5.
;
;--
TD_INTERRUPT:                                   ; Service device interrupt
        MOVL    @(SP)+,R4                       ; Get address of IDB and
                                                ; remove pointer from stack.
        ASSUME  IDB$L_CSR  EQ  0
        ASSUME  IDB$L_OWNER  EQ  4
        MOVQ    IDB$L_CSR(R4),R4                ; Get address of device's CSR
                                                ; Get address of device
                                                ; owner's UCB.
        DEVICELOCK  LOCKADDR=UCB$L_DLCK(R5), -  ; Lock device access
                PRESERVE=NO,-                   ; Don't preserve R0
                CONDITION=NOSETIPL              ; Don't bother setting IPL
        BBCC    #UCB$V_INT,-                    ; If device does not expect
                UCB$W_STS(R5),-                 ; interrupt, dismiss it.
                UNSOL_INTERRUPT

;
; This is a solicited interrupt. Save
; the contents of the device registers in the UCB.
;
```

```
        MOVW    TD_STATUS(R4),-         ; Otherwise, save all device
                UCB_W_TD_STATUS(R5)     ; registers. First the CSR.
        MOVW    TD_WRDCNT(R4),-         ; Save the word count register.
                UCB_W_TD_WRDCNT(R5)
        MOVW    TD_BUFADR(R4),-         ; Save the buffer address
                UCB_W_TD_BUFADR(R5)     ; register.
        MOVW    TD_DATBUF(R4),-         ; Save the data buffer register.
                UCB_W_TD_DATBUF(R5)

;
; Restore control to the main driver.
;
RESTORE_DRIVER:                         ; Jump to main driver code.
        MOVL    UCB$L_FR3(R5),R3        ; Restore driver's R3 (use a
                                        ; MOVQ to restore R3-R4).
        JSB     @UCB$L_FPC(R5)          ; Call driver at interrupt
                                        ; wait address.

;
; Dismiss the interrupt.
;
UNSOL_INTERRUPT:                        ; Dismiss unsolicited interrupt.
        DEVICEUNLOCK LOCKADDR=UCB$L_DLCK(R5),- ; Unlock device access.
                PRESERVE=NO             ; Don't bother preserving R0.
        POPR    #^M<R0,R1,R2,R3,R4,R5>  ; Restore R0-R5
        REI                             ; Return from interrupt.


        .SBTTL  TD_CANCEL, Cancel I/O routine

;++
; TD_CANCEL, Cancels an I/O operation in progress
;
; Functional description:
;
;       This routine calls IOC$CANCELIO to set the cancel bit in the
;       UCB status longword if:
;
;               the device is busy,
;               the IRP's process ID matches the cancel process ID,
;               the IRP channel matches the cancel channel.
;
;       If IOC$CANCELIO sets the cancel bit, then this driver routine
;       does device-dependent cancel I/O fixups.
;
; Inputs:
;
;       R2      - channel index number
;       R3      - address of the current IRP (I/O request packet)
;       R4      - address of the PCB (process control block) for the
;                 process canceling I/O
;       R5      - address of the UCB (unit control block)
;       R8      - cancel reason code, one of:
;                 CAN$C_CANCEL    if called through $CANCEL system service
;                 CAN$C_DASSGN    if called through $DASSGN or
;                                 $DALLOC system services
;                 These reason codes are defined by the $CANDEF macro.
;
; Outputs:
;
;       The routine must preserve all registers except R0-R3.
;
;       The routine may set the UCB$M_CANCEL bit in UCB$W_STS.
```

```
;
;--
TD_CANCEL:                              ; Cancel an I/O operation
        JSB     G^IOC$CANCELIO          ; Set cancel bit if appropriate.
        BBC     #UCB$V_CANCEL,-         ; If the cancel bit is not set,
                UCB$W_STS(R5),10$       ; just return.

;
; Device-dependent cancel operations go next.
;

;
; Finally, the return.
;
10$:
        RSB                             ; Return

        .SBTTL  TD_REG_DUMP, Device register dump routine

;++
; TD_REG_DUMP, Dumps the contents of device registers to a buffer
;
; Functional description:
;
;       Writes the number of device registers, and their current
;       contents into a diagnostic or error buffer.
;
; Inputs:
;
;       R0      - address of the output buffer
;       R4      - address of the CSR (controller status register)
;       R5      - address of the UCB (unit control block)
;
; Outputs:
;
;       The routine must preserve all registers except R1-R3.
;
;       The output buffer contains the current contents of the device
;       registers. R0 contains the address of the next empty longword in
;       the output buffer.
;
;--
TD_REG_DUMP:                            ; Dump device registers
        MOVZBL  #TD_NUM_REGS,(R0)+      ; Store device register count.
        MOVZWL  UCB_W_TD_STATUS(R5),-   ; Store device status register.
                (R0)+
        MOVZWL  UCB_W_TD_WRDCNT(R5),-   ; Store word count register.
                (R0)+
        MOVZWL  UCB_W_TD_BUFADR(R5),-   ; Store buffer address register.
                (R0)+
        MOVZWL  UCB_W_TD_DATBUF(R5),-   ; Store data buffer register.
                (R0)+
        RSB                             ; Return

        .SBTTL  TD_END, End of driver

;++
; Label that marks the end of the driver
;--
TD_END:                                 ; Last location in driver
        .END
```

B VMS SCSI Class Driver Template

This appendix lists the contents of the VMS SCSI class driver template. The code in this template can serve as the starting point for a new third-party SCSI class driver. You can obtain a machine-readable copy of this driver from SYS$EXAMPLES:SKDRIVER.MAR.

```
        .TITLE  SKDRIVER - VAX/VMS Sample SCSI Class Driver
        .IDENT  'X-3'
;       .LIST   MEB

;*****************************************************************************
;*                                                                           *
;*  COPYRIGHT (c) 1989 BY                                                    *
;*  DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASSACHUSETTS.                   *
;*  ALL RIGHTS RESERVED.                                                     *
;*                                                                           *
;*  THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED AND COPIED    *
;*  ONLY IN  ACCORDANCE WITH  THE  TERMS  OF  SUCH  LICENSE  AND WITH THE    *
;*  INCLUSION OF THE ABOVE COPYRIGHT NOTICE. THIS SOFTWARE OR  ANY  OTHER    *
;*  COPIES THEREOF MAY NOT BE PROVIDED OR OTHERWISE MADE AVAILABLE TO ANY    *
;*  OTHER PERSON.  NO TITLE TO AND OWNERSHIP OF  THE  SOFTWARE IS  HEREBY    *
;*  TRANSFERRED.                                                             *
;*                                                                           *
;*  THE INFORMATION IN THIS SOFTWARE IS  SUBJECT TO CHANGE WITHOUT NOTICE    *
;*  AND  SHOULD  NOT  BE  CONSTRUED AS  A COMMITMENT BY DIGITAL EQUIPMENT    *
;*  CORPORATION.                                                             *
;*                                                                           *
;*  DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE  OR  RELIABILITY OF ITS    *
;*  SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DIGITAL.                  *
;*                                                                           *
;*                                                                           *
;*****************************************************************************
;

;++
;
; FACILITY:
;
;       VAX/VMS Sample SCSI Class Driver
;
; ABSTRACT:
;
;       This module contains a sample SCSI class driver. This template
;       supports two modes of operation: either the SCSI command
;       packets are formatted in the application program (passthru mode) or
;       the SCSI command packets are formatted within the driver. In the
;       latter case, command  processing and error recovery are implemented
;       within a third-party SCSI class driver derived from this driver.
;
;       Passthru mode is the method of access used by the generic SCSI
;       class driver (GKDRIVER). GKDRIVER provides access to a SCSI device
;       from an application program. The QIO interface of the GKDRIVER
;       is fixed. However, third-party SCSI  class drivers can define a
;       unique QIO interface. Third-party  class drivers can have device
;       specific error recovery, log device errors and implement asynchro-
;       nous event notification (AEN). Third-party  class drivers have
;       direct access to the SCSI Port Interface (SPI) routines, while
```

```
;       using the passthru function provides access to SCSI without writing
;       a driver.
;
;       The code to perform the IO$_DIAGNOSE function is included in this
;       driver for informational purposes only. Typical third-party  SCSI
;       class drivers do not require this function. If the IO$_DIAGNOSE
;       function is required, you should use the VMS-supported  SCSI
;       generic class driver (GKDRIVER).
;
;       SKDRIVER supports three I/O functions:
;
;               IO$_AVAILABLE   - Inquiry and Test Unit Ready sequence,
;               IO$_DIAGNOSE    - Passthru function
;               IO$_READLBLK    - Return Inquiry data to user
;
;


        .SBTTL  +
        .SBTTL  + SYMBOL DEFINITIONS
        .SBTTL  +
        .SBTTL  External symbol definitions
;+
; External symbols
;-

        $CRBDEF                         ; Channel request block
        $DDBDEF                         ; Device data block
        $DEVDEF                         ; Device characteristics
        $EMBDEF                         ; Error log message buffer
        $DYNDEF                         ; Data structure types
        $FKBDEF                         ; Define fork block symbols
        $IODEF                          ; I/O function codes
        $IPLDEF                         ; Hardware IPL definitions
        $IRPDEF                         ; I/O request packet
        $PCBDEF                         ; Process control block
        $PRVDEF                         ; Privilege mask
        $SCDRPDEF                       ; SCSI SCDRP symbols
        $SSDEF                          ; System status codes
        $UCBDEF                         ; Unit control block
        $VECDEF                         ; Interrupt vector block


        .SBTTL  Miscellaneous local symbols
;+
; Local symbols
;-

;+
; Argument list (AP) offsets for device-dependent QIO parameters
;-
P1      = 0                             ; First QIO parameter
P2      = 4                             ; Second QIO parameter
P3      = 8                             ; Third QIO parameter
P4      = 12                            ; Fourth QIO parameter
P5      = 16                            ; Fifth QIO parameter
P6      = 20                            ; Sixth QIO parameter
```

```
SCDRPS_PER_UNIT         = 2             ; Number of SCDRPs to allocate per
                                        ; unit
UCB_STACK_SIZE          = 10            ; Size of internal stack in UCB
MAX_BCNT                = ^XFFFF        ; Maximum byte count
.IIF NDF DT$_GENERIC_SCSI, DT$_GENERIC_SCSI     = 5 ; GENERIC SCSI DEVICE
ASSEMBLE_PASSTHRU       = 0             ; If 0 don't assemble DIAG code, if
                                        ; 1 do.
SCSI$M_STS              = ^XC1          ; Used to extract vendor unique STS
                                        ; bits.
DIAG_BUF_LEN            = 60            ; Length in bytes of DIAGNOSE input
                                        ; buffer.
MAX_CMD_LEN             = 248           ; Maximum size in bytes of a SCSI
                                        ; CMD.
INQ_DATA_LEN            = 36            ; Exact number of INQUIRY bytes
                                        ; required.
NUM_ARGS                = 10            ; Number of SET/GET CONNECTION CHAR
                                        ; arguments.

        .SBTTL  SCSI Peripheral Device Types

;+
; Define SCSI Peripheral Device Types
;-
        SCSI_C_DA               = 0     ; Direct Access
        SCSI_C_SA               = 1     ; Sequential Access
        SCSI_C_PT               = 2     ; Printer
        SCSI_C_PR               = 3     ; Processor
        SCSI_C_WR               = 4     ; Write-once Read-multiple
        SCSI_C_RO               = 5     ; Read-only direct access


        .SBTTL  Sense key codes

;+
; Define SCSI sense key codes.
;-
        SCSI_C_NO_SENSE         = 0     ; No sense data
        SCSI_C_RECOVERED_ERROR  = 1     ; Recovered error (treated as
                                        ; success)
        SCSI_C_NOT_READY        = 2     ; Device not ready
        SCSI_C_MEDIUM_ERROR     = 3     ; Medium (parity) error
        SCSI_C_HARDWARE_ERROR   = 4     ; Hardware error
        SCSI_C_ILLEGAL_REQUEST  = 5     ; Illegal request
        SCSI_C_UNIT_ATTENTION   = 6     ; Unit attention (media change,
                                        ; reset)
        SCSI_C_DATA_PROTECT     = 7     ; Data protection (write lock error)
        SCSI_C_BLANK_CHECK      = 8     ; Blank check (advance past end of
                                        ; data)
        SCSI_C_VENDOR_UNIQUE    = 9     ; Vendor unique key
        SCSI_C_COPY_ABORTED     = 10    ; Copy operation aborted
        SCSI_C_ABORTED_COMMAND  = 11    ; Command aborted
        SCSI_C_EQUAL            = 12    ; Compare operation, data match
        SCSI_C_VOLUME_OVERFLOW  = 13    ; Write beyond physical end of tape
        SCSI_C_MISCOMPARE       = 14    ; Compare operation, data mismatch


;++
; Define offsets in various SCSI command packets.
;--
```

```
;+
; REQUEST SENSE data offsets.
;-
        SCSI_XS_B_ERR_CODE      = 0     ; Extended sense error code
        SCSI_XS_B_KEY           = 2     ; Extended sense KEY field
        SCSI_XS_V_KEY           = 0     ; Extended sense KEY bit number
        SCSI_XS_S_KEY           = 4     ; Extended sense KEY length
        SCSI_XS_B_ADDNL_INFO    = 3     ; Extended sense additional code
        SCSI_XS_B_ADDNL_CODE    = 12    ; Extended sense additional code
        SCSI_XS_B_ADDNL_CODE30  = 8     ; "                       " (TZ30)
        SCSI_XS_B_ADDNL_CODE50  = 8     ; "                       " (TZK50)
        SCSI_XS_M_EOF           = ^X80  ; Extended sense end of file
        SCSI_XS_M_EOM           = ^X40  ; Extended sense end of medium
        SCSI_XS_M_ILI           = ^X20  ; Extended sense illegal length
                                        ; indicator
        SCSI_XS_V_ADDNL_VALID   = 7     ; Extended sense additional data
                                        ; valid

;+
; INQUIRY data offsets.
;-
        SCSI_INQ_B_DEVTYPE      = 0     ; Inquiry device type
        SCSI_INQ_B_DEVQUAL      = 1     ; Inquiry device qualifier field
        SCSI_INQ_V_DEVQUAL      = 0     ; Inquiry device qualifier starting
                                        ; bit
        SCSI_INQ_S_DEVQUAL      = 7     ; Inquiry device qualifier length
        SCSI_INQ_V_REMOVABLE    = 7     ; Inquiry removable bit

        SCSI_SKIP_B_CNT         = 2     ; Skip record count

;+
; MODE SELECT/SENSE data offsets.
;-
        SCSI_MSNS_B_WP          = 2     ; Mode sense write protect field
        SCSI_MSNS_V_WP          = 7     ; Mode sense write protect bit

        SCSI_MSEL_W_RSVD0       = 0     ; Mode select reserved
        SCSI_MSEL_B_SPEED       = 2     ; Mode select speed field
        SCSI_MSEL_B_MODE        = 2     ; Mode select buffered mode
        SCSI_MSEL_B_DSCLEN      = 3     ; Mode select record descriptor
                                        ; length
        SCSI_MSEL_C_DSCLEN      = 8     ; Mode select record descriptor
                                        ; length
        SCSI_MSEL_B_DENS        = 4     ; Mode select density
        SCSI_MSEL_B_BLOCKS      = 5     ; Mode select number of blocks
        SCSI_MSEL_B_RSVD1       = 8     ; Mode select reserved
        SCSI_MSEL_B_BLKLEN      = 9     ; Mode select block length
        SCSI_MSEL_B_VULEN       = 12    ; Mode select vendor unique length
        SCSI_MSEL_B_VU          = 13    ; Mode select vendor unique field
        SCSI_MSEL_M_BUF         = ^X10  ; Mode select buffered mode
        SCSI_MSEL_M_NOF         = 7     ; Number of fillers for generic
                                        ; device
        SCSI_MSEL_M_NOF50       = 7     ; Number of fillers for TZK50
        SCSI_MSEL_M_NOF30       = ^X0F  ; Number of fillers for TZ30
        SCSI_MSEL_M_RESEL       = ^X40  ; Reselection timeout flag
;+
; SPI interface, Get/set connect characteristics symbols.
;-
        SET_CON_L_LEN           = 0     ; Length field
        SET_CON_L_CON_FLAGS     = 4     ; Flags field
        SET_CON_M_DISC          = 1     ; Enable disconnect flag
        SET_CON_M_NORETRY       = 2     ; Disable command retry flag
        SET_CON_L_SYN_FLAG      = 8     ; Synchronous flag field
        SET_CON_M_SYN           = 1     ; Synchronous flag
```

```
        .SBTTL  Template class driver extensions to the UCB
;+
; Template class driver extensions to the UCB.
;-
        $DEFINI UCB                         ; Start of UCB definitions

        .=UCB$K_LCL_DISK_LENGTH             ; Position at end of UCB

$DEF    UCB_L_STACK_PTR .BLKL   1           ; Internal stack pointer
$DEF    UCB_L_STACK     .BLKL   UCB_STACK_SIZE  ; Internal stack
$DEF    UCB_L_SCDRP     .BLKL   1           ; Address of active SCDRP
$DEF    UCB_L_SCDT      .BLKL   1           ; SCDT address
$DEF    UCB_L_SK_FLAGS  .BLKL   1           ; Class driver flags
        _VIELD  UCB,0,<-                ;
                <DISCONNECT,,M>,-       ; Device supports disconnect
                <DISABL_ERRLOG,,M>,-    ; Disable error logging
                <SYNCHRONOUS,,M>>       ; Device supports synchro-
                                                ; nous operation
$DEF    UCB_W_PHASE_TMO .BLKW   1           ; Phase change timeout
$DEF    UCB_W_DISC_TMO .BLKW    1           ; Disconnect timeout
$DEF    UCB_L_SCDRPQ_FL .BLKL   1           ; Queue of free SCDRPs used to
$DEF    UCB_L_SCDRPQ_BL .BLKL   1           ; send SCSI commands

$DEF    UCB_L_SAVER6    .BLKL   1           ; Safe place for R6.
$DEF    UCB_L_SAVER7    .BLKL   1           ; Safe place for R7.
$DEF    UCB_L_SCDRP_SAV1 .BLKL  1           ; Safe place for SCDRP address.
$DEF    UCB_B_LUN       .BLKB   1           ; Save device LUN

$DEF    UCB_K_SK_UCBLEN                     ; Length of extended UCB

        $DEFEND UCB                         ; End of UCB definitions

        .SBTTL  Error log packet formats
;+
; The following are the definitions for class driver error log packets.
; The VMS error log formatter formats third-party  SCSI class driver
; error log packets. The ERF utility formats a standard error
; log packet for third-party  class drivers. The standard packet is defined
; below. If a user would like to dump additional data to the error log,
; simply increase the size of the error log packet defined. The additional
; data will be dumped as untranslated longwords in the error log.
;-
        $DEFINI ERROR_PACKETS

        . = EMB$L_DV_REGSAV                 ; Start of area to dump error info

$DEF    ERR_LW_CNT      .BLKL   1           ; Count of number of longwords that
                                            ; follow
$DEF    ERR_REVISION    .BLKB   1           ; Revision level
$DEF    ERR_HW_REV      .BLKL   1           ; Hardware revision
$DEF    ERR_TYPE        .BLKB   1           ; Error type
$DEF    ERR_SCSI_ID     .BLKB   1           ; SCSI ID
$DEF    ERR_SCSI_LUN    .BLKB   1           ; SCSI logical unit
$DEF    ERR_SCSI_SUBLUN .BLKB   1           ; SCSI sublogical unit
$DEF    ERR_PORT_STATUS .BLKL   1           ; Port status code
$DEF    ERR_CMD_LEN     .BLKB   1           ; SCSI command length field
$DEF    ERR_CMD_BYTES   .BLKB   12          ; Maximum possible command bytes
$DEF    ERR_SCSI_STS    .BLKB   1           ; SCSI status byte
$DEF    ERR_TXT_LEN     .BLKB   1           ; Error message text size
$DEF    ERR_TXT_BYTES   .BLKB   60          ; Maximum possible text bytes
        .=.+4                               ; Reserve one longword after end of
                                            ; defined packet.
$DEF    ERR_K_COMMAND_LENGTH                ; Length of packet containing SCSI
                                            ; command

        $DEFEND ERROR_PACKETS
```

```
        .SBTTL  SCSI Class driver error log types.
;+
; SCSI class driver error log types. Each error that is logged by the
; class driver should have a unique error type.
;-
CLS_DRV_ERROR_01 = 1                    ; Class driver specific error type.
CLS_DRV_ERROR_02 = 2                    ; Class driver specific error type.
CLS_DRV_ERROR_03 = 3                    ; Class driver specific error type.
CLS_DRV_ERROR_04 = 4                    ; Class driver specific error type.
CLS_DRV_ERROR_05 = 5                    ; Class driver specific error type.
CLS_DRV_ERROR_06 = 6                    ; Class driver specific error type.

        .SBTTL  +
        .SBTTL  + MACRO DEFINITIONS
        .SBTTL  +
        .SBTTL  SCSI_CMD        - Define a SCSI command packet
;+
; SCSI_CMD
;
; This macro defines the contents of a SCSI command packet. Each SCSI com-
; mand can have associated with it a DMA buffer used during the
; DATAIN/DATAOUT bus phases. A DMA length of zero indicates there is no
; DATA(IN/OUT) phase associated with this command (except in the case of a
; read/write SCSI command, which is handled specially.)
; Class drivers can specify on a command by command basis the DMA Timeout
; and Disconnect Timeout values. The disconnect timeout is the maximum
; number of seconds that an I/O can be disconnected from the bus. A timeout
; of -1 allows an infinite timeout. The DMA timeout is the maximum timeout
; for a DMA transfer to complete or a phase change on the SCSI bus to occur;
; this timeout is also in units of seconds.
; The SETUP_CMD routine uses this information in preparing to send a SCSI
; command. The macro generates a label and the SCSI command information as
; follows:
;
;       +-----------------------+
;       |    SCSI cmd length    | 1 byte
;       +-----------------------+
;       |    SCSI cmd bytes     | n bytes
;       +-----------------------+
;       |   DMA buffer length   | 2 bytes
;       +-----------------------+
;       |     DMA direction     | 1 byte
;       +-----------------------+
;       |     DMA Timeout       | 1 longword
;       +-----------------------+
;       |  Disconnect Timeout   | 1 longword
;       +-----------------------+
;
;
; DMA direction is defined as: 0=write, 1=read.
;-
        .MACRO  SCSI_CMD, NAME, CMD_BYTES, DMA_LEN=0, DMA_DIR=READ,-
                          DMA_TMO=0, DISCON_TMO=0
'NAME'_CMD:
        $$$BYTE_COUNT=0
        .IRP CMD_BYTE, <CMD_BYTES>
        $$$BYTE_COUNT = $$$BYTE_COUNT + 1
        .IIF EQ $$$BYTE_COUNT-1, SCSI_C_'NAME' = CMD_BYTE    ; Define opcode
        .ENDR
        .BYTE   $$$BYTE_COUNT
        .IRP CMD_BYTE, <CMD_BYTES>
        .BYTE   CMD_BYTE
        .ENDR
```

```
        .WORD   DMA_LEN
        $$$DIRECTION = 0
        .IIF IDN DMA_DIR, READ, $$$DIRECTION = 1
        .BYTE   $$$DIRECTION
        .LONG   DMA_TMO
        .LONG   DISCON_TMO
        .ENDM   SCSI_CMD

        .SBTTL  LOG_ERROR       - Log a SCSI class driver error
;+
; LOG_ERROR
;
; This macro logs a SCSI class driver error. The error type and VMS status
; code are placed in R7 and R8 respectively, and the LOG_ERROR routine is
; called.
;-

        .MACRO  LOG_ERROR,TYPE,VMS_STATUS,UCB=R3,MESSAGE='',?LABEL_1

        .SHOW EXPANSIONS
        PUSHR   #^M<R5,R7,R8,R11>       ; Save registers
        .IF DIF UCB,R5
        MOVL    UCB,R5                  ; Get UCB address
        .ENDC
        MOVL    #'TYPE',R7              ; Get error code
        MOVL    VMS_STATUS,R8           ; And VMS status code
        .IF LESS_THAN   60-%LENGTH(MESSAGE) ; Maximum size message is 60
        .ERROR ; Message text is greater than 60 characters
        .ENDC
        .SAVE_PSECT LOCAL_BLOCK
        .PSECT  $$$111_TEXT
        LABEL_1:
        .ASCIC  /'MESSAGE'/
        .RESTORE_PSECT
        MOVAL   LABEL_1,R11
        BSBW    LOG_ERROR               ; Write an error log entry
        POPR    #^M<R5,R7,R8,R11>       ; Restore registers
        .NOSHOW EXPANSIONS

        .ENDM   LOG_ERROR

        .SBTTL  WORD_BRANCHES   - Define word displacement branches

;+
; WORD_BRANCHES
;
; This macro defines for each Bxxx (conditional branch) instruction an
; equivalent macro named BxxxW with a word displacement. The macro takes
; as an argument a list of tuples, each tuple containing 3 items: 1) a
; conditional branch opcode; 2) the opcode with the opposite polarity;
; and 3) the number of arguments required by the opcode.
;-
        .MACRO  WORD_BRANCHES LIST

        .MACRO  WORD_BRANCHES2, OPCODE1, OPCODE2, ARGCNT

        .IF EQ ARGCNT-0
        .MACRO  OPCODE1, DST, ?L
        OPCODE2 L
        BRW     DST
L:      .ENDM   OPCODE1
        .ENDC
```

```
        .IF EQ ARGCNT-1
        .MACRO  OPCODE1, FIELD, DST, ?L
        OPCODE2 FIELD,L
        BRW     DST
L:      .ENDM   OPCODE1
        .ENDC

        .IF EQ ARGCNT-2
        .MACRO  OPCODE1, BIT, FIELD, DST, ?L
        OPCODE2 BIT,FIELD,L
        BRW     DST
L:      .ENDM   OPCODE1
        .ENDC

        .ENDM   WORD_BRANCHES2

        .MACRO  WORD_BRANCHES1, OPCODE1, OPCODE2, ARGCNT

        WORD_BRANCHES2 'OPCODE1'W, OPCODE2, ARGCNT
        WORD_BRANCHES2 'OPCODE2'W, OPCODE1, ARGCNT

        .ENDM   WORD_BRANCHES1

        .IRP    ENTRY, <LIST>
        WORD_BRANCHES1 ENTRY
        .ENDR

        .ENDM   WORD_BRANCHES

        WORD_BRANCHES <-
                <BBC,   BBS,    2>,-
                <BBCC,  BBSC,   2>,-
                <BBCS,  BBSS,   2>,-
                <BCC,   BCS,    0>,-
                <BEQL,  BNEQ,   0>,-
                <BEQLU, BNEQU,  0>,-
                <BGEQ,  BLSS,   0>,-
                <BGEQU, BLSSU,  0>,-
                <BGTR,  BLEQ,   0>,-
                <BGTRU, BLEQU,  0>,-
                <BLBC,  BLBS,   1>,-
                <BVC,   BVS,    0>>

        .SBTTL  INIT_UCB_STACK  - Initialize the internal UCB stack
        .SBTTL  SUBPUSH         - Push an item on the UCB stack
        .SBTTL  SUBPOP          - Pop an item from the UCB stack
        .SBTTL  SUBSAVE         - Save a return address on the UCB stack
        .SBTTL  SUBRETURN       - Return to the address saved on UCB stack
;+
; INIT_UCB_STACK
; SUBPUSH
; SUBPOP
; SUBSAVE
; SUBRETURN
;
; These macros manipulate the UCB internal stack, which is used to save
; routine return address and temporary variables.
;-
        .MACRO  INIT_UCB_STACK,UCB=R5,?L1

        MOVAL   UCB_L_STACK-4(UCB),-
                UCB_L_STACK_PTR(UCB)

        .ENDM   INIT_UCB_STACK

        .MACRO  SUBPUSH,ARG,UCB=R3,?L1,?L2
```

```
        ADDL    #4,UCB_L_STACK_PTR(UCB)
        MOVL    ARG,@UCB_L_STACK_PTR(UCB)

        .ENDM   SUBPUSH

        .MACRO  SUBPOP,ARG,UCB=R3,?L1,?L2

        MOVL    @UCB_L_STACK_PTR(UCB),ARG
        SUBL    #4,UCB_L_STACK_PTR(UCB)

        .ENDM   SUBPOP

        .MACRO  SUBSAVE,UCB=R3,?L1,?L2

        SUBPUSH (SP)+,UCB

        .ENDM   SUBSAVE

        .MACRO  SUBRETURN,UCB=R3,?L1,?L2

        SUBPOP  -(SP),UCB
        RSB

        .ENDM   SUBRETURN


        .SBTTL  SK_WAIT    - Stall a thread for a specific number of seconds
;+
; SK_WAIT
;
; This macro uses the device timeout mechanism to stall a thread for a
; specified number of seconds. The UCB address and stall time are required
; as inputs.
;-
        .MACRO  SK_WAIT,SECONDS,UCB=R5,SCRATCH=R0,?L

        .IF DIF UCB,R5
        MOVL    R5,SCRATCH
        MOVL    UCB,R5
        MOVL    SCRATCH,UCB
        .ENDC
        DSBINT  ENVIRON=UNIPROCESSOR
        PUSHL   SECONDS
        BSBW    SK_WAIT
        .WORD   L-.
L:      IOFORK
        BICW    #UCB$M_TIMOUT,-
                UCB$W_STS(R5)
        .IF DIF UCB,R5
        MOVL    UCB,SCRATCH
        MOVL    R5,UCB
        MOVL    SCRATCH,R5
        .ENDC

        .ENDM   SK_WAIT

        .SBTTL  +
        .SBTTL  + DRIVER TABLES
        .SBTTL  +
        .SBTTL  Driver prologue table
;+
; Driver prologue table
;
; This table provides various information about the driver, such as its name
; and length, and causes initialization of various fields in the I/O
; database when the driver is loaded.
;-
        .IIF NDF DPT$M_NO_IDB_DISPATCH, DPT$M_NO_IDB_DISPATCH = ^X1000
```

```
        DPTAB   -                                       ; DPT-creation macro
                END=SK_END,-                            ; End of driver label
                ADAPTER=NULL,-                          ; Adapter type
                UCBSIZE=<UCB_K_SK_UCBLEN>,-     ; Length of UCB
                NAME=SKDRIVER,-                         ; Driver name
                FLAGS=<DPT$M_SMPMOD!-   ; Driver runs in SMP
                                                        ; environment
                      DPT$M_NO_IDB_DISPATCH>    ; Don't fill in IDB$L_UCBLST
        DPT_STORE INIT                                  ; Start of load
                                                        ; initialization table
        DPT_STORE UCB,UCB$L_MAXBCNT,L,MAX_BCNT  ; Maximum byte count
        DPT_STORE UCB,UCB$B_FLCK,B,SPL$C_IOLOCK8 ; Device FORK LOCK
        DPT_STORE UCB,UCB$B_DIPL,B,22           ; Device interrupt IPL
        DPT_STORE UCB,UCB$L_DEVCHAR,L,<-  ; Device characterstics
                DEV$M_AVL!-                             ; Available
                DEV$M_IDV!-                             ; Input device
                DEV$M_ODV!-                             ; Output device
                DEV$M_SHR!-                             ; Shareable Device
                DEV$M_ELG!-                             ; Error logging enabled
                DEV$M_RND>                              ; Random Access Device
        DPT_STORE UCB,UCB$L_DEVCHAR2,L,<- ; Device characterstics
                DEV$M_NNM>                              ; Prefix name with "node$"
        DPT_STORE UCB,UCB$B_DEVTYPE,B,DT$_GENERIC_SCSI ; Generic SCSI device
        DPT_STORE UCB,UCB$B_DEVCLASS,B,DC$_MISC ; Sample device class
        DPT_STORE UCB,UCB$W_DEVSTS,W,-          ; Set no logical to physical
                  UCB$M_NOCNVRT                 ; block number conversion
        DPT_STORE REINIT                                ; Start of reload
                                                        ; initialization table
        DPT_STORE DDB,DDB$L_DDT,D,SK$DDT        ; Address of DDT
        DPT_STORE CRB,-                                 ; Address of controller
                CRB$L_INTD+VEC$L_INITIAL,-      ; initialization routine
                D,SK_CTRL_INIT
        DPT_STORE CRB,-                                 ; Address of device
                CRB$L_INTD+VEC$L_UNITINIT,-     ; unit initialization
                D,SK_UNIT_INIT                  ; routine
        DPT_STORE CRB,CRB$B_FLCK,B,IPL$_IOLOCK8 ; Initialize fork lock field

        DPT_STORE END                                   ; End of initialization
                                                        ; tables

        .SBTTL  Driver dispatch table
;+
; Driver dispatch table
;
; This table defines the entry points into the driver.
;-

        DDTAB   -                                       ; DDT-creation macro
                DEVNAM=SK,-                             ; Name of device
                START=SK_STARTIO,-                      ; Start I/O routine
                FUNCTB=SK_FUNCTABLE,-                   ; FDT address
                REGDMP=SK_REG_DUMP                      ; Register dump routine

        .SBTTL  Function decision table
;+
; Function decision table
;
; This table lists the $QIO function codes implemented by the driver and
; the preprocessing routines used by each function.
;-
```

```
SK_FUNCTABLE:                                   ; FDT for driver
        FUNCTAB,-                               ; Valid I/O functions
                 <AVAILABLE,-       ; Inquiry and Test Unit Ready
                  READLBLK,-                    ; Perform a "read" function
                  READVBLK,-                    ; Perform a "read" function
                  DIAGNOSE>                     ; Special pass-through function

        FUNCTAB,<>                  ; Buffered I/O functions

        FUNCTAB SK_READ,<READLBLK,READVBLK>  ; Issue SCSI
                                                        ; INQUIRY command
        FUNCTAB +EXE$ZEROPARM,<AVAILABLE>    ; Issue SCSI INQUIRY
                                                        ; command
        FUNCTAB SK_DIAGNOSE,<DIAGNOSE>       ; Special pass-
                                                        ; through function


        .SBTTL  SCSI Command Packet Definition Table

SK_CMD_DEFS::
        SCSI_CMD -
                NAME = TEST_UNIT_READY,-
                CMD_BYTES = <0, 0, 0, 0, 0, 0>

        SCSI_CMD NAME = INQUIRY,-
                CMD_BYTES = <18 , 0, 0, 0, 36, 0>,-
                DMA_LEN = 36,-
                DMA_DIR = READ,-
                DMA_TMO = 0,-                   ; Use default
                DISCON_TMO = 0                  ; Use default

        SCSI_CMD NAME = REQUEST_SENSE,-
                CMD_BYTES = <3, 0, 0, 0, 18, 0>,-
                DMA_LEN = 18,-
                DMA_DIR = READ,-
                DMA_TMO = 0,-                   ; Use default
                DISCON_TMO = 0                  ; Use default

        SCSI_CMD NAME = MODE_SELECT,-
                CMD_BYTES = <21, 0, 0, 0, 4, 0>,-
                DMA_LEN = 4,-
                DMA_DIR = WRITE,-
                DMA_TMO = 0,-                   ; Use default
                DISCON_TMO = 0                  ; Use default

        SCSI_CMD NAME = QIO_INQUIRY,-           ; Normally this would be
                CMD_BYTES = <18 , 0, 0, 0, 0, 0>,- ; a read/write
                                                        ; command
                DMA_LEN = -1,-                  ; If data goes to user
                DMA_DIR = READ,-                ; buffer, then use -1 here.
                DMA_TMO = 0,-                   ; Use default
                DISCON_TMO = 0                  ; Use default

SK_CMD_DEFS_END =.
```

```
        .SBTTL  +
        .SBTTL  + DRIVER ENTRY POINTS
        .SBTTL  +
        .SBTTL  SK_CTRL_INIT    - Controller initialization routine
;++
; SK_CTRL_INIT
;
; This routine is called to perform controller-specific initialization
; and is called by the operating system in three places:
;
;       - at system startup
;       - during driver loading and reloading
;       - during recovery from a power failure
;
; Currently this routine is a NOP.
;
; INPUTS:
;
;       R4      - address of the CSR (controller status register)
;       R5      - address of the IDB (interrupt data block)
;       R6      - address of the DDB (device data block)
;       R8      - address of the CRB (channel request block)
;
; OUTPUTS:
;
;       All registers preserved
;--
SK_CTRL_INIT:
        MOVZWL  #SS$_NORMAL,R0          ; Set success status
        RSB                             ; Return to caller


        .SBTTL  SK_UNIT_INIT    - Unit initialization routine
;++
; SK_UNIT_INIT
;
; This routine allocates a set of SCDRPs and places them on a queue in the
; UCB, forms a connection to the port driver by calling SPI$CONNECT, and
; sets the unit online.
;
; INPUTS:
;
;       R5      - UCB address
;
; OUTPUTS:
;
;       R0-R3   - Destroyed
;       All other registers preserved
;--
SK_UNIT_INIT:                           ; Initialize unit
        BBC     #UCB$V_POWER,-          ; Branch if we're not here due to a
                UCB$W_STS(R5),2$        ; powerfail
        RSB                             ; Otherwise, exit immediately

;+
; Fork twice for now to allow the port driver's unit init routine to execute
; before ours.
;-
2$:     FORK                    ; Fork to drop IPL to SYNCH
        FORK                    ; 2nd Fork synchronizes with port driver.
        INIT_UCB_STACK          ; Initialize the internal stack in the UCB
```

```
        MOVAL   UCB_L_SCDRPQ_FL(R5),R0  ; Initialize the SCDRP queue header
        MOVL    R0,(R0)                 ; in the UCB
        MOVL    R0,4(R0)                ;
        MOVL    #SCDRPS_PER_UNIT,R4     ; Number of SCDRPs allocated per unit
10$:    MOVL    #<SCDRP$C_LENGTH>,R1    ; Length of SCDRP
        MOVL    R5,R3                   ; Copy UCB address
        BSBW    ALLOC_POOL              ; Go allocate an SCDRP
        MOVW    R1,SCDRP$W_SCDRPSIZE(R2); Save length of SCDRP
        INSQUE  SCDRP$L_FQFL(R2),-      ; Place SCDRP in UCB queue
                UCB_L_SCDRPQ_FL(R5)     ;
        SOBGTR  R4,10$                  ; Repeat for all SCDRPs

;+
; All SCSI device unit numbers should be of the form "n0m" where n is the
; SCSI ID between 0 and 7 and m is the LUN between 0 and 7. Extract the ID
; from the LUN by dividing the unit number by 100. The quotient is then
; used as the ID while the remainder is the LUN. Note that the unit number
; contains three digits because early versions of SCSI provided for sub-
; logical unit numbers. This feature has since been removed and the second
; digit in the unit number is not used.
;-
        MOVZWL  #SS$_BADPARAM,R0        ; Assume bad LUN or SUBLUN specified
        MOVZWL  UCB$W_UNIT(R5),R1       ; Get device unit number
        CLRL    R2                      ; Prepare for extended divide
        EDIV    #100,R1,R1,R2           ; Extract SCSI bus ID from LUN
        CMPL    R1,#7                   ; Valid SCSI ID (0 <= n
                                        ; <= 7)?
        BGTRUW  20$                     ; Branch if not
        CMPL    R2,#7                   ; Valid LUN (0 <= n
                                        ; <= 7)?
        BGTRUW  20$                     ; Branch if not
        MULB3   #<1@5>,R2,UCB_B_LUN(R5)  ; Save LUN (shifted left
                                        ; 5 bits for use later in SETUP_CMD)
        ASHL    #16,R1,R1               ; Place SCSI ID in high-order word
                                        ; of R1
        ASHL    #16,R2,R2               ; Place LUN in high-order word of R2
        MOVL    UCB$L_DDB(R5),R0        ; Get DDB address
        SUBB3   #^A'A',-                ; Translate controller letter to
                DDB$T_NAME+3(R0),R1     ; SCSI bus ID
        SPI$CONNECT                     ; Connect to the port driver
        BLBC    R0,20$                  ; Branch if connect attempt failed
        CMPL    R1,UCB$L_MAXBCNT(R5)    ; For MAXBCNT, use minimum supported
        BGEQ    15$                     ; value of port and class drivers
        MOVL    R1,UCB$L_MAXBCNT(R5)    ; Save maximum byte count in UCB
15$:    MOVL    R2,UCB_L_SCDT(R5)       ; Save SCDT address
        MOVL    R4,UCB$L_PDT(R5)        ; Save PDT address

        BISW    #UCB$M_ONLINE,-         ; Set unit online
                UCB$W_STS(R5)           ;
20$:    RSB                             ; Return to caller

        .SBTTL  +
        .SBTTL  + QIO FDT INTERFACE ROUTINES
        .SBTTL  +
```

```
        .SBTTL  SK_READ  - FDT preprocessing for sending SCSI Inquiry command
;++
; SK_READ
;
; This routine performs FDT preprocessing including:
;
;       - Validating access to, and locking, the read/write buffer
;
; INPUTS:
;
;       R0      - Address of FDT routine
;       R3      - IRP address
;       R4      - PCB address
;       R5      - UCB address
;       R6      - CCB address
;       R7      - Bit number of user-specified I/O function code
;       R8      - Address of current entry in FDT
;       AP      - Address of first function-dependent argument (P1)
;
; OUTPUTS:
;
;--
SK_READ:

;+
; Use system routines to execute I/O preprocessing.
;-
        TSTL    P2(AP)                  ; There must be bytes to receive.
        BEQL    BADPARAM                ; Bad input parameters.
        JMP     G^EXE$MODIFY            ; Lock down pages, set up IRP,
                                        ; JUMP to EXE$QIODRVPKT, etc...

BADPARAM:
        MOVZWL  #SS$_BADPARAM,R0        ; Set bad parameter status
        JMP     G^EXE$ABORTIO           ; Abort the I/O with status in R0

        .SBTTL  SK_DIAGNOSE  - FDT preprocessing for special pass-through
                               function
;++
; SK_DIAGNOSE
;
; This routine performs FDT preprocessing including:
;
;       - Validating access to the descriptor buffer
;       - Validating access to, and locking, the read/write buffer
;       - Copying the SCSI command to a buffer in nonpaged pool
;
; INPUTS:
;
;       R0      - Address of FDT routine
;       R3      - IRP address
;       R4      - PCB address
;       R5      - UCB address
;       R6      - CCB address
;       R7      - Bit number of user-specified I/O function code
;       R8      - Address of current entry in FDT
;       AP      - Address of first function-dependent argument (P1)
;
; OUTPUTS:
;
;--
```

```
        DSC_OPCODE = 0
        DSC_FLAGS = 4
        DSC_CMDADR = 8
        DSC_CMDLEN = 12
        DSC_DATADR = 16
        DSC_DATLEN = 20
        DSC_PADCNT = 24
        DSC_PHSTMO = 28
        DSC_DSCTMO = 32
SK_DIAGNOSE:
        .IF NOT_EQUAL ASSEMBLE_PASSTHRU ; Flag to control assembly of
                                        ; IO$_DIAGNOSE
        IFPRIV  DIAGNOSE,10$            ; Branch if process has DIAGNOSE priv
        MOVZWL  #SS$_NOPRIV,R0          ; Set no privilege status
        BRW     50$                     ; Branch to abort the I/O
;+
; First, check that we have read access to the user's descriptor.
;-
10$:    MOVQ    (AP),R0                 ; Get user descriptor address, length
        MOVL    R0,R9                   ; Save a copy of descriptor address
        CMPL    R1,#DIAG_BUF_LEN        ; Valid descriptor length
        BLSSW   40$                     ; Branch if not
        JSB     G^EXE$WRITECHK          ; Check for read access to the
                                        ; descriptor buffer (don't return if
                                        ; no access)

        CMPL    DSC_OPCODE(R9),#1       ; Valid opcode?
        BNEQW   40$                     ; Branch if not

        CMPL    DSC_DATLEN(R9),-        ; Reasonable read/write data buffer
                UCB$L_MAXBCNT(R5)       ; length?
        BGTRUW  40$                     ; Branch if not
        CMPL    DSC_PADCNT(R9),#511     ; Reasonable pad count?
        BGTRU   40$                     ; Branch if not

        MOVQ    DSC_CMDADR(R9),R0       ; Get SCSI command buffer address,
                                        ; length
        CMPL    R1,#MAX_CMD_LEN         ; Valid command length?
        BGTRU   40$                     ; Branch if not
        JSB     G^EXE$WRITECHK          ; Check for read access to the command
                                        ; buffer (don't return if no access)
        ADDL    #8,R1                   ; Reserve space for command buffer
                                        ; overhead
        JSB     G^EXE$ALONONPAGED       ; Allocate a buffer in which to copy
                                        ; the SCSI command
        BLBC    R0,50$                  ; Branch on error
        MOVL    R1,(R2)+                ; Save length of buffer
        MOVL    R2,IRP$L_MEDIA(R3)      ; Save the command buffer address
        MOVL    DSC_CMDLEN(R9),R0       ; Get length of the SCSI command
        MOVL    R0,(R2)+                ; Save it in the command buffer
        PUSHR   #^M<R2,R3,R4,R5>        ; Save registers
        MOVC3   R0,@DSC_CMDADR(R9),(R2) ; Copy the SCSI command from user's
                                        ; buffer to the buffer in pool
        POPR    #^M<R2,R3,R4,R5>        ; Restore registers
        CLRL    IRP$L_BCNT(R3)          ; Assume no user read/write data
        MOVL    DSC_DATADR(R9),R0       ; Get address of user data buffer
        BEQL    30$                     ; Branch if no user read/write data
        MOVL    DSC_DATLEN(R9),R1       ; Get length of user data buffer
        BEQL    30$                     ; Branch if no user read/write data
        MOVAL   G^EXE$READLOCKR,R2      ; Assume user is performing a read
        BLBS    DSC_FLAGS(R9),20$       ; Branch if this is a read operation
        MOVAL   G^EXE$WRITE LOCKR,R2    ; Other check for read access
20$:    JSB     (R2)                    ; Check access to and lock down buffer
        BLBC    R0,60$                  ; Branch on error
```

```
30$:    MOVAL   IRP$C_CDRP(R3),R0       ; Get address of SCDRP within IRP
        MOVL    DSC_FLAGS(R9),(R0)+     ; Save flags field in IRP/SCDRP
        MOVAL   DSC_PADCNT(R9),R1       ; Get address of pad count field
        .REPT   3
        MOVL    (R1)+,(R0)+             ; Save pad count, timeout values
        .ENDR
        JMP     G^EXE$QIODRVPKT         ; Queue the packet to the driver

40$:    MOVZWL  #SS$_BADPARAM,R0        ; Set bad parameter status
50$:    JMP     G^EXE$ABORTIO           ; Abort the I/O with status in R0

;+
; We arrive here if the last FDT operation - checking access to and locking
; down the user's read/write buffer - fails. EXE$READLOCKR or EXE$WRITE LOCKR
; returns to us through a coroutine call to allow us to give up any resources
; which we have allocated during FDT processing. Deallocate the buffer
; containing a copy of the SCSI command, then return from the coroutine call.
; R0 and R1 must be preserved.
;-
60$:    PUSHQ   R0                      ; Save registers
        MOVL    IRP$L_MEDIA(R3),R0      ; Get address of nonpaged pool buffer
                                        ; containing SCSI command
        MOVL    -(R0),R1                ; Get length of buffer
        JSB     G^EXE$DEANONPGDSIZ      ; Deallocate the packet
        POPQ    R0                      ; Restore registers
        RSB                             ; Return from coroutine call
        .ENDC                           ; IF ASSEMBLE_PASTHRU

        .IF EQUAL ASSEMBLE_PASSTHRU     ; IF IO$_DIAGNOSE not assembled, do
                                        ; this..
        MOVZBL  #SS$_ILLIOFUNC,R0       ; Specify the error type
        JMP     G^EXE$ABORTIO           ; Abort the I/O with status in R0
        .ENDC

        .SBTTL  +
        .SBTTL  + STARTIO SCSI COMMAND EXECUTION ROUTINES
        .SBTTL  +

        .SBTTL  SK_STARTIO      - Driver STARTIO entry point
;++
; SK_STARTIO
;
; This routine is the STARTIO entry point into the driver. Its main function
; is to dispatch to the function-code-specific routine that starts a specific
; I/O function.
;
; INPUTS:
;
;       R3      - IRP address
;       R5      - UCB address
;
; OUTPUTS:
;
;       R0      - 1st longword of I/O status: contains status code and
;                 number of bytes transferred
;       R1      - 2nd longword of I/O status: low-order word contains
;                 high-order word of number of bytes transferred
;       R4      - Destroyed
;       All other registers preserved
;--

SK_STARTIO:
        .ENABLE LSB                     ; SK_STARTIO

        INIT_UCB_STACK                  ; Initialize the internal stack in the UCB
```

```
        MOVL    UCB$L_PDT(R5),R4        ; Get PDT address
        MOVL    R3,R2                   ; Copy IRP address
        MOVL    R5,R3                   ; Copy UCB address
        BSBW    ALLOC_SCDRP             ; Allocate an SCDRP
        MOVL    R2,SCDRP$L_IRP(R5)      ; Save IRP address in SCDRP

        EXTZV   #IRP$V_FCODE,-          ; Extract I/O function code
                #IRP$S_FCODE,-          ;
                IRP$W_FUNC(R2),R1       ;
        ASSUME  IRP$S_FCODE LE 7        ; Allow byte mode dispatch
        DISPATCH R1,TYPE=B,<-    ; Dispatch according to function
                <IO$_DIAGNOSE,  IO_DIAGNOSE>,-
                <IO$_READPBLK,  IO_READ>,-
                <IO$_AVAILABLE, IO_INQUIRY>>
;+
; Bogus I/O function code will fall through. Set illegal function code
; status and complete the I/O.
;-
IO_BOGUS:
        MOVZBL  #SS$_ILLIOFUNC,R0       ; Specify the error type
;       BRB     COMPLETE_IO             ; Fall through to exit path for
                                        ; if other error then uncomment.

COMPLETE_IO:
        BSBW    DEALLOC_SCDRP           ; Deallocate the SCDRP
        MOVL    R3,R5                   ; Copy UCB address
        REQCOM                          ; Complete the I/O
        .DISABLE LSB                    ; SK_STARTIO

        .SBTTL  IO_INQUIRY      - Send SCSI INQUIRY command.
;++
; IO_INQUIRY
;
; This routine is intended as an example of how to write a STARTIO
; routine for a SCSI class driver.
;
; This routine sends an inquiry command to the target. If
; errors occur during the execution of this operation no retries
; occur. However, this class driver issues a REQUEST SENSE to
; determine the nature of the event. If the event is fatal, the
; error is logged and the I/O fails. If the event is
; benign, then the I/O completes with a REQCOM.
;
; IO_INQUIRY calls the port driver to allocate command buffer areas,
; maps the system or user buffer such that the port driver has access
; to these areas, and then calls the port driver's SEND_CMD entry point
; to send the SCSI command to a target.
; When the port driver returns from this call, the INQUIRY data has been
; moved, the command status is in the status-in buffer and the SCSI
; bus is free. The class driver checks the transfer count, releases
; its resources and completes the I/O with a call to REQCOM.
;
; INPUTS:
;
;       R3      - UCB address
;       R4      - PDT address
;       R5      - SCDRP address
;
; OUPUTS:
;
;       R0      - Status
;
;                       SS$_NORMAL    - I/O completed successfully.
;                       SS$_ILLSEQOP - I/O failed, bad sense key.
```

```
;                       SS$_IVSTSFLG - Invalid SCSI status returned.
;                       SS$_OPINCMPL - I/O failed, insufficient data returned.
;
;--

IO_INQUIRY:
        .ENABLE LSB                         ; IO_INQUIRY
        MOVAL   INQUIRY_CMD,R2              ; Address of INQUIRY command
        BSBW    SETUP_CMD                   ; Perform setup for SCSI command
        BLBC    R0,35$                      ;
        BSBW    SEND_COMMAND                ; Send the SCSI command
;+
; Determine by sending the INQUIRY command, what target is at this ID.
;
; After a call to the port driver, when the port status (R0) and SCSI
; command status have been checked, the class driver must verify that
; the number of bytes that were to be received or sent have been delivered
; by the port driver. SCDRP$L_TRANS_CNT contains the actual number of bytes
; of data transferred by the port driver.
;-
        BLBC    R0,35$                      ; Branch on error
        CMPL    SCDRP$L_TRANS_CNT(R5),-     ; Sufficient inquiry data returned?
                #INQ_DATA_LEN               ;
        BLSSUW  34$                         ; Branch if not
        MOVL    SCDRP$L_SVA_USER(R5),R1     ; Get address of inquiry data
        CMPB    #SCSI_C_DA,-                ; Is this a SCSI disk device?
                SCSI_INQ_B_DEVTYPE(R1)      ; Check INQUIRY data
;***    BNEQ    SOMEWHERE                   ; If it's not the target you want.
30$:    BSBW    CLEANUP_CMD                 ; Clean up from the SCSI command
;+
; Now that the class driver knows what target is out there, determine if
; the target is ready by sending a TEST UNIT READY command.
;-
        MOVAL   TEST_UNIT_READY_CMD,R2      ; Test Unit Ready command
        BSBW    SETUP_CMD                   ; Perform setup for SCSI command
        BLBC    R0,35$                      ; Branch on error
        BSBW    SEND_COMMAND                ; Send the SCSI command
        BLBC    R0,35$                      ; Branch on error
        BSBW    CLEANUP_CMD                 ; Clean up from the SCSI command
        CLRL    R1                          ; Clean up R1
        BRW     COMPLETE_IO                 ; Complete the user's I/O.
;+
; Any error the class driver encounters is logged.
; R0 contains the VMS status.
;-
34$:    MOVZWL  #SS$_OPINCOMPL,R0
35$:    LOG_ERROR -                         ; Log an invalid inquiry data error
                TYPE=CLS_DRV_ERROR_01,-     ;
                VMS_STATUS=R0,-             ; I/O operation failed
                UCB=R3,-                    ;
                MESSAGE=<ERROR DURING INQUIRY_TEST UNIT RDY
                SEQUENCE>
        BSBW    CLEANUP_CMD                 ; Clean up from the SCSI command
        CLRL    R1                          ; Clean up R1
        BRW     COMPLETE_IO                 ; Complete the user's I/O.
        .DISABLE LSB                        ; IO_INQUIRY
```

```
        .SBTTL  IO_READ - Send SCSI INQUIRY command and return data.
;++
; IO_READ
;
; This routine is intended as an example of how to write a STARTIO
; routine that reads data from a target device and returns the data
; to a user buffer. Normally, some form of read command would be used
; to retrieve data from a target; however the format of read commands
; varies depending on the SCSI device class. Therefore, this
; example uses the INQUIRY command to get data from the target; the
; INQUIRY command is one of the few commands that is common among
; all device types.
;
; Third-party class drivers traditionally do NOT return the INQUIRY
; data to the application. Rather, the class driver uses this
; information to establish the characteristics of the SCSI target
; and the class driver's connection to this target.
;
; IO_READ calls the port driver to allocate command buffer areas,
; maps user read buffer such that the port driver has access to these
; areas and then calls the port driver's SEND_CMD entry point
; to send the SCSI command to a target. When the port driver returns from
; this call, the INQUIRY data has been moved to the user's buffer,
; the command status is in the status-in buffer and the SCSI bus is free.
; The class driver checks the transfer count, releases its resources and
; complete the I/O with a call to REQCOM.
;
; INPUTS:
;
;       R3      - UCB address
;       R4      - PDT address
;       R5      - SCDRP address
;
; OUPUTS:
;
;       R0      - Status
;
;                     SS$_NORMAL   - I/O completed successfully.
;                     SS$_ILLSEQOP - I/O failed, bad sense key.
;                     SS$_IVSTSFLG - Invalid SCSI status returned.
;                     SS$_OPINCMPL - I/O failed, insufficient data returned
;
;--

IO_READ:
        .ENABLE LSB                     ; IO_READ
```

```
;+
; WARNING: If the user provides the wrong byte count the SCSI bus may hang.
; SCSI port drivers can recover from this error; however, the recovery
; mechanism may be severe and this I/O request will fail.
;-
        MOVL    #SCDRP$M_BUFFER_MAPPED,- ; Set buffer mapped flag to prevent
                SCDRP$L_SCSI_FLAGS(R5)  ; allocation of S0 buffer for data
        MOVAL   QIO_INQUIRY_CMD,R2      ; Address of INQUIRY command for
                                        ; user data
        BSBW    SETUP_CMD               ; Perform setup for SCSI command
        BLBC    R0,300$                 ; Setup failed
        SPI$MAP_BUFFER                  ; Map the user buffer
        BSBW    SEND_COMMAND            ; Send the SCSI command
;+
; The port driver has been called to send the command and now returns
; with the data moved to the user's buffer, the port status in R0, and SCSI
; status in the STATUSIN buffer. The class driver checks the port driver
; and SCSI command status and then verifies that the number of bytes that
; were  received equals the BCNT. SCDRP$L_TRANS_CNT contains the actual
; number of bytes of data transferred by the port driver.
;-
        BLBC    R0,35$                  ; Branch on error
        CMPL    SCDRP$L_TRANS_CNT(R5),- ; Sufficient inquiry data returned?
                SCDRP$L_BCNT(R5)
        BNEQUW  34$                     ; Branch if not
30$:    MOVL    SCDRP$L_TRANS_CNT(R5),R1; Return transaction count in IOSB
        BSBW    CLEANUP_CMD             ; Clean up from the SCSI command
        BRW     COMPLETE_IO             ; Complete the user's I/O

;+
; Errors the class driver encounters are logged.
; R0 contains the VMS status.
;-
34$:    MOVZWL  #SS$_OPINCOMPL,R0
35$:    LOG_ERROR -                     ; Log an invalid inquiry data error
                TYPE=CLS_DRV_ERROR_04,- ;
                VMS_STATUS=R0,-         ; I/O operation failed
                UCB=R3,-                        ;
                MESSAGE=<ERROR DURING READ QIO FUNCTION>
        BSBW    CLEANUP_CMD             ; Clean up from the SCSI command
        CLRL    R1                      ; Clean up R1
        BRW     COMPLETE_IO             ; Complete the user's I/O

;+
; The template driver does not support segmented I/O. This exercise
; is left to the user.
;-
300$:   BICL    #SCDRP$M_BUFFER_MAPPED,-; No buffer mapped, so don't unmap.
                SCDRP$L_SCSI_FLAGS(R5)  ;
        LOG_ERROR -                     ; Log an invalid inquiry data error
              TYPE=CLS_DRV_ERROR_05,- ;
              VMS_STATUS=R0,-         ; I/O operation failed.
              UCB=R3,-                        ;
              MESSAGE=<ERROR I_O OPERATION NOT PROPERLY SEGMENTED>
        CLRL    R1                      ; Clean up R1
        BRW     COMPLETE_IO             ; Complete the user's I/O
        .DISABLE LSB                    ; IO_READ
```

```
        .SBTTL  IO_DIAGNOSE     - Special pass-through function
;++
; IO_DIAGNOSE
;
; STARTIO routine for the passthru function of the template SCSI
; class driver. This routine assumes that the user has provided
; a buffer that contains the SCSI command packet and that the
; FDT routines in the driver have made the appropriate checks
; during I/O preprocessing to allow access to the user data areas
; during STARTIO.
;
; IO_DIAGNOSE makes calls into the port driver to allocate command
; buffer areas, maps the user buffer such that the port driver
; can access user areas, and then calls the port driver's SEND_CMD
; entry point to send the SCSI command to a target. When the port
; driver returns from this call, the user's data has been moved,
; the command status is in the status-in buffer and the SCSI bus
; is free. The class driver releases its resources and
; completes the I/O with a call to REQCOM.
;
; INPUTS:
;
;       R2      - IRP address
;       R3      - UCB address
;       R4      - PDT address
;       R5      - SCDRP address
;
; OUTPUTS:
;
;       R0      - Status
;       R1,R2   - Destroyed
;       All other registers preserved
;--

IO_DIAGNOSE:
        .ENABLE LSB                         ; IO_DIAGNOSE
        .IF NOT_EQUAL ASSEMBLE_PASSTHRU ; IF assemble IO$_DIAGNOSE if
                                            ; ASSM_PASS
        MOVL    IRP$L_MEDIA(R2),-           ; Copy command buffer from IRP to
                SCDRP$L_MEDIA(R5)           ; SCDRP
        MOVL    IRP$L_SVAPTE(R2),-          ; and SVAPTE,
                SCDRP$L_SVAPTE(R5)          ;
        MOVL    IRP$L_BCNT(R2),-            ; BCNT,
                SCDRP$L_BCNT(R5)            ;
        MOVW    IRP$W_BOFF(R2),-            ; and BOFF
                SCDRP$W_BOFF(R5)            ;
        MOVW    IRP$W_STS(R2),-             ; and STS
                SCDRP$W_STS(R5)             ;
        MOVAL   IRP$C_CDRP(R2),R0           ; Get address of SCDRP portion of IRP
        EXTZV   #1,#1,(R0),R1               ; Get disconnect flag
        INSV    R1,#UCB_V_DISCONNECT,-      ; Fill in disconnect flag in UCB
                #1,UCB_L_SK_FLAGS(R3)       ;
        EXTZV   #2,#1,(R0),R1               ; Get synchronous flag
        INSV    R1,#UCB_V_SYNCHRONOUS,-     ; Fill in synchronous flag in UCB
                #1,UCB_L_SK_FLAGS(R3)       ;
        ADDL    #4,R0                       ; Advance to pad count field
        MOVL    (R0)+,-                     ; Fill in the pad count in the SCDRP
                SCDRP$L_PAD_BCNT(R5)        ;
        MOVL    (R0)+,-                     ; Fill in the phase change (DMA)
                                            ; timeout
                SCDRP$L_DMA_TIMEOUT(R5)     ; in the SCDRP
        MOVL    (R0)+,-                     ; Fill in the disconnect timeout in
                SCDRP$L_DISCON_TIMEOUT(R5) ; the SCDRP
        BSBW    SET_CONN_CHAR               ; Set up the connect characteristics
```

```
        MOVL    SCDRP$L_MEDIA(R5),R1    ; Get address of SCSI command in pool
        MOVL    (R1)+,R1                ; Get length of SCSI command
        ADDL    #8,R1                   ; Account for overhead
        SPI$ALLOCATE_COMMAND_BUFFER     ; Allocate a command buffer
        MOVL    R2,SCDRP$L_CMD_BUF(R5)  ; Save address of command buffer
        CLRL    (R2)+                   ; Reserve a longword for status
        MOVB    #^XFF,-1(R2)            ; Initialize status field
        MOVAL   -1(R2),-                ; Address to save status byte
                SCDRP$L_STS_PTR(R5)     ;
        MOVL    R2,SCDRP$L_CMD_PTR(R5)  ; Address of SCSI command in cmd
                                        ; buffer
        MOVL    SCDRP$L_MEDIA(R5),R0    ; Get SCSI command in pool again
        MOVL    (R0),(R2)+              ; Copy SCSI command length
        PUSHR   #^M<R0,R2,R3,R4,R5>     ; Save registers
        MOVC3   (R0),4(R0),(R2)         ; Copy SCSI command to command buffer
        POPR    #^M<R0,R2,R3,R4,R5>     ; Restore registers
        MOVL    -(R0),R1                ; Get length of command buffer in pool
        JSB     G^EXE$DEANONPGDSIZ      ; Deallocate the buffer
        TSTL    SCDRP$L_BCNT(R5)        ; Any user data buffer?
        BEQL    10$                     ; Branch if not
        SPI$MAP_BUFFER                  ; Map the user's data buffer
10$:    SPI$SEND_COMMAND                ; Send the SCSI command
        PUSHL   R0                      ; Save returned port status
        TSTL    SCDRP$L_BCNT(R5)        ; User buffer mapped?
        BEQL    20$                     ; Branch if not
        SPI$UNMAP_BUFFER                ; Unmap the user's data buffer
20$:    MOVL    SCDRP$L_CMD_BUF(R5),R0  ; Get the command buffer address
        PUSHL   (R0)                    ; Save the SCSI status byte
        SPI$DEALLOCATE_COMMAND_BUFFER   ; Deallocate the command buffer
        POPL    R1                      ; Restore the SCSI status byte
        POPL    R0                      ; Restore the port status
        INSV    SCDRP$L_TRANS_CNT(R5),- ; Copy the transfer count to the
                #16,#16,R0              ; high-order word of R0
        .ENDC                           ; If ASS_DIAG FALSE don't assemble
        BRW     COMPLETE_IO             ; Complete the QIO
        .DISABLE LSB                    ; IO_DIAGNOSE

        .SBTTL  +
        .SBTTL  + UTILITY ROUTINES
        .SBTTL  +

        .SBTTL  SEND_COMMAND    - Send a SCSI command
;++
; SEND_COMMAND
;
; This routines sends a command to the SCSI device. It returns any failing
; port status to the caller. If the port status is success, it checks the
; SCSI status byte. If a check condition status is returned, a request
; sense command is sent to the target and the sense key is translated into
; a VMS status code, which is returned as status.
;
; INPUTS:
;
;       R3      - UCB address
;       R4      - PDT address
;       R5      - SCDRP address
;
; OUTPUTS:
;
;       R0      - Status
;                       SS$_IVSTSFLG - Invalid SCSI status returned.
;                       SS$_ILLSEQOP - I/O operation failed.
;       R1,R2   - Destroyed
```

```
;       All other registers preserved
;--
SEND_COMMAND:
        .ENABLE LSB                         ; SEND_COMMAND
        SUBSAVE                             ; Save return address
        SPI$SEND_COMMAND                    ; Send the SCSI command
        BLBC    R0,10$                      ; If port failed, return
        MOVZBL  @SCDRP$L_STS_PTR(R5),R1 ; Get SCSI status byte
        BICB    #SCSI$M_STS,R1              ; Clear reserved, vendor-unique bits
        BNEQ    20$                         ; Branch if bad status
10$:    SUBRETURN                           ; Return to caller

;+
; A bad SCSI status code was returned. If the code is a check condition,
; then send a request sense command to the device. Otherwise, the status
; code is something unexpected. Log an error and return SS$_MEDOFL status.
;-
20$:    CMPB    R1,#2                       ; Check condition status?
        BNEQ    90$                         ; Branch if not
;+
; A check condition status code was returned. Save the original SCDRP
; address, allocate a second one and send a request sense command. If the
; request sense succeeds, translate the sense key to a VMS status code and
; return that as the status code for the original command.
;-
45$:    MOVL    R5,UCB_L_SCDRP_SAV1(R3) ; Save original SCDRP address
        BSBW    ALLOC_SCDRP             ; Allocate an additional SCDRP
        BSBW    REQUEST_SENSE           ; Send a request sense command
        BLBC    R0,50$                  ; Branch on error
                                        ; a VMS status code in R0
;+
; Look at the results of the request sense to determine the exact nature
; of the event.
;-
        MOVL    SCDRP$L_SVA_USER(R5),R1 ; Get address of REQUEST SENSE DATA.
        BICB3   #^XF0,SCSI_XS_B_ERR_CODE(R1),- ; First check ERROR CODE.
                R0                      ; In this case zero is good, but this
        BNEQ    50$                     ; is really device specific.
        BICB3   #^XF0,SCSI_XS_B_KEY(R1),- ; Mask off SENSE KEY.
                R0
;+
; Depending on the value of the sense key, dispatch to the appropriate
; error recovery.
;-
        DISPATCH R0,TYPE=B,<-                   ; Dispatch accord-
                                                ; ing to SENSE KEY.
                <SCSI_C_NO_SENSE,SK_OK>,-       ; No sense data
                <SCSI_C_RECOVERED_ERROR,SK_OK>,- ; Recovered error
                <SCSI_C_NOT_READY,SK_BAD>,-     ; Device not ready
                <SCSI_C_MEDIUM_ERROR,SK_BAD>,-  ; Medium (parity)
                                                ; error
                <SCSI_C_HARDWARE_ERROR,SK_BAD>,-  ; Hardware error
                <SCSI_C_ILLEGAL_REQUEST,SK_BAD>,- ; Illegal request
                <SCSI_C_UNIT_ATTENTION,SK_BAD>,-  ; Unit attention
                                                  ; ( reset... )
                <SCSI_C_DATA_PROTECT,SK_BAD>,-  ; Data protection
                                                ; (write lock)
                <SCSI_C_BLANK_CHECK,SK_BAD>,-   ; Blank check
                <SCSI_C_VENDOR_UNIQUE,SK_BAD>,- ; Vendor unique key
                <SCSI_C_COPY_ABORTED,SK_BAD>,-  ; Copy operation
                                                ; aborted
                <SCSI_C_ABORTED_COMMAND,SK_BAD>,- ; Command aborted
```

```
                <SCSI_C_EQUAL,SK_BAD>,-         ; Data match
                <SCSI_C_VOLUME_OVERFLOW,SK_BAD>,- ; Write past
                                                        ; physical end
                <SCSI_C_MISCOMPARE,SK_BAD>>     ; Data mismatch

;+
; Either the sense key was bad or the key was invalid. In either case
; indicate that the command failed. Some class drivers will want
; to translate each bad sense key to a unique class driver SS$_XXXXX
; status code. Here we will always return SS$_ILLSEQOP.
;-
SK_BAD:
        MOVL    #SS$_ILLSEQOP,R0        ; I/O operation failed
        BRB     50$                     ; cleanup and return error
;+
; If the sense key indicated that the operation completed successfully,
; then return success.
;-
SK_OK:
        MOVL    #SS$_NORMAL,R0          ; I/O operation succeeded
;       BRB     50$                     ; Clean up and return error

50$:    BSBW    CLEANUP_CMD             ; Clean up the request sense command
        BSBW    DEALLOC_SCDRP           ; Deallocate the request sense SCDRP
        MOVL    UCB_L_SCDRP_SAV1(R3),R5 ; Restore original SCDRP address
        MOVL    R5,UCB_L_SCDRP(R3)      ; Copy it to the UCB
        BRW     10$                     ; Return to caller

;+
; If the status returned for the last command was anything other than
; check condition, log an error and return a status of SS$_IVSTSFLG to
; indicate that command failed and that there is no request sense data.
;-
90$:    MOVL    #SS$_IVSTSFLG,R0        ; Return a generic status code
        LOG_ERROR -                     ; Log a send command error
                TYPE=CLS_DRV_ERROR_02,- ; Generic user class driver error
                VMS_STATUS=R0,-         ;
                UCB=R3,-                ;
                MESSAGE=<ERROR BAD SCSI COMMAND STATUS>
        BRW     10$
        .DISABLE LSB                    ; SEND_COMMAND

        .SBTTL  REQUEST_SENSE   - Send a request sense command
;++
; REQUEST_SENSE
;
; This routine is called by SEND_COMMAND when a command fails with check
; condition status. A request sense command is sent to the target.
;
; INPUTS:
;
;       R3      - UCB address
;       R4      - PDT address
;       R5      - SCDRP address
;
; OUTPUTS:
;
;       R0      - Status
;                       SS$_IVSTSFLG - Bad SCSI status returned during
;                                      REQUEST SENSE.
;       R1,R2   - Destroyed
;       All other registers preserved
;--
```

```
REQUEST_SENSE:
        .ENABLE LSB                         ; REQUEST_SENSE
        SUBSAVE                             ; Save return address
        MOVAL   REQUEST_SENSE_CMD,R2        ; Address of REQUEST_SENSE command
        BSBW    SETUP_CMD                   ; Perform setup for SCSI command
        BLBC    R0,10$                      ; Branch on error
        SPI$SEND_COMMAND                    ; Send the SCSI command
        BLBC    R0,10$                      ; Return on error
        MOVZBL  @SCDRP$L_STS_PTR(R5),R1     ; Get SCSI status byte
        BICB    #SCSI$M_STS,R1              ; Clear reserved, vendor unique bits
        BNEQ    20$                         ; Branch if bad status
10$:    SUBRETURN                           ; Return to caller

20$:    MOVZWL  #SS$_IVSTSFLG,R0            ; Return bad SCSI status to caller.
        BRB     10$
        .DISABLE LSB                        ; REQUEST_SENSE
        .SBTTL  SET_CONN_CHAR   - Modify connection characteristics
;++
; SET_CONN_CHAR
;
; This routine is called to initialize the connection characteristics,
; which  specify such things as whether the device supports disconnect
; and synchronous operation, and the bus busy, arbitration, selection,
; and command retry counters.
;
; This routine first does a SPI$GET_CONNECTION_CHAR to get the current
; values of the connection characteristics, modifies the values of interest,
; then does a SPI$SET_CONNECTION_CHAR to set up the new values. This allows
; the class driver to change a subset of the characteristics and leave the
; rest unmodified.
;
; INPUTS:
;
;       R3      - UCB address
;       R4      - SPDT address
;       R5      - SCDRP address
;
; OUTPUTS:
;
;       R0-R2   - Destroyed
;       All other registers preserved
;--

SET_CONN_CHAR:
        .ENABLE LSB                         ; SET_CONN_CHAR
```

```
        SUBSAVE                         ; Save return address
        MOVL    #<<NUM_ARGS+1>*4>,R1    ; Size of
                                        ; get/set connection char buffer
        BSBW    ALLOC_POOL              ; Allocate the buffer
        SUBPUSH R2                      ; Save address of buffer
        MOVL    #NUM_ARGS,(R2)          ; Set argument count in buffer
        SPI$GET_CONNECTION_CHAR         ; Get current connection
                                        ; characteristics
        BLBC    R0,10$                  ; Branch on error
;+
; Some devices won't select if selected with attention.
;
; NOTE: It is strongly suggested that targets and devices
;       support the disconnect/reselection sequence. All
;       Digital-supplied devices support this feature to
;       ensure consistent bus performance.
;
;       EXTZV   #UCB_V_DISCONNECT,#1,-  ; Fill in disconnect flag
;               UCB_L_SK_FLAGS(R3),4(R2);
;-
        EXTZV   #UCB_V_SYNCHRONOUS,#1,-   ; Fill in synchronous flag
                UCB_L_SK_FLAGS(R3),8(R2)  ;
        SPI$SET_CONNECTION_CHAR         ; Set the connection characteristics
10$:    PUSHL   R0                      ; Save return status
        SUBPOP  R0                      ; Get address of characteristics
                                        ; buffer
        BSBW    DEALLOC_POOL            ; Deallocate the buffer
        POPL    R0                      ; Restore return status
        BLBS    R0,20$                  ; Branch if success status
        MOVL    #SS$_CTRLERR,R0         ; Otherwise, return a reasonable
                                        ; status
20$:    SUBRETURN                       ; Return to caller
        .DISABLE LSB                    ; SET_CONN_CHAR

        .SBTTL  SK_WAIT         - Stall for the specified number of seconds
;++
; SK_WAIT
;
; This routine is used by the SK_WAIT macro to stall a thread for a
; specified number of seconds. It sets the timeout bit in the UCB and
; relies on the device timeout mechanism to resume the stalled thread.
;
; INPUTS:
;
;       IPL     - 31
;       R5      - UCB address
;       (SP)    - Return address
;       4(SP)   - Wait time in seconds
;       8(SP)   - Saved IPL
;       12(SP)  - Address of caller's caller
;
; OUPUTS:
;
;       Stack   - Return address, wait time, IPL removed
;       Control returns to caller's caller
;       All registers preserved
;
;       NOTE: The use of the SK_WAIT macro destroys R0-R3
;--
SK_WAIT:
```

```
        MOVQ    R3,UCB$L_FR3(R5)        ; Save R3 and R4 in fork block
        ADDL3   #2,(SP)+,UCB$L_FPC(R5)  ; Save return address in fork block
        BISW    #UCB$M_TIM,UCB$W_STS(R5); Set timer expected bit
        ADDL3   (SP)+,G^EXE$GL_ABSTIM,- ; Set up timeout time in UCB
                UCB$L_DUETIM(R5)        ;
        BICW    #UCB$M_TIMOUT,-         ; Clear timer expired bit
                UCB$W_STS(R5)           ;
        ENBINT                          ; Reenable interrupts
        RSB                             ; Return to caller's caller

        .SBTTL  ALLOC_SCDRP     - Allocate an SCDRP
;++
; ALLOC_SCDRP
;
; This routine allocates an SCDRP by attempting to remove one from the
; queue in the UCB. If the queue is empty (which should never happen),
; then bugcheck. The entire SCDRP is zeroed and various fields are
; initialized.
;
; INPUTS:
;
;       R3      - UCB address
;
;       UCB_L_SCDRPQ_FL - Queue of SCDRPs
;
; OUTPUTS:
;
;       R5      - SCDRP address
;       All other registers preserved
;
;       SCDRP$L_UCB     - UCB address
;       SCDRP$L_IRP     - IRP address
;       SCDRP$L_CDT     - SCDT address
;       SCDRP$L_SCSI_FLAGS - Initialized
;       SCDRP$L_CL_SSK_PTR - Initialized
;--
ALLOC_SCDRP:
        .ENABLE LSB                     ; ALLOC_SCDRP
        REMQUE  @UCB_L_SCDRPQ_FL(R3),R5 ; Remove an SCDRP from the queue
        PUSHR   #^M<R0,R1,R2,R3,R4,R5>  ; Save registers
        MOVC5   #0,.,#0,-               ; Initialize the SCDRP
                #SCDRP$C_LENGTH-12,-    ;
                12(R5)                  ;
        POPR    #^M<R0,R1,R2,R3,R4,R5>  ; Restore registers
        MOVL    R5,UCB_L_SCDRP(R3)      ; Save SCDRP address in UCB
        MOVL    R3,SCDRP$L_UCB(R5)      ; Save UCB address in SCDRP
        MOVB    UCB$B_FLCK(R3),-        ; Copy the fork lock field from the
                SCDRP$B_FLCK(R5)        ; UCB to the SCDRP
        MOVL    UCB_L_SCDT(R3),-        ; Save SCDT address in SCDRP
                SCDRP$L_CDT(R5)         ;
        MOVAL   SCDRP$L_SCSI_STK-4(R5),-; Initialize the SCDRP stack pointer
                SCDRP$L_SCSI_STK_PTR(R5);
        RSB

        .DISABLE LSB                    ; ALLOC_SCDRP
```

```
        .SBTTL  DEALLOC_SCDRP   - Deallocate an SCDRP
;++
; DEALLOC_SCDRP
;
; This routine deallocates an SCDRP by returning it to the queue in the
; UCB. A sanity check is made to ensure that any map registers for this
; command have been deallocated.
;
; INPUTS:
;
;       R3      - UCB address
;       R5      - SCDRP address
;
; OUTPUTS:
;
;       R3      - UCB address
;       R5      - UCB address (for _R5 entry point)
;       All other registers preserved
;
;       UCB_L_SCDRP     - Cleared to indicate no active SCDRP
;--

DEALLOC_SCDRP:

        .ENABLE LSB                     ; DEALLOC_SCDRP
        INSQUE  SCDRP$L_FQFL(R5),-      ; Insert SCDRP in UCB queue
                UCB_L_SCDRPQ_FL(R3)     ;
        CLRL    UCB_L_SCDRP(R3)         ; No active SCDRP for this UCB
        RSB

        .DISABLE LSB                    ; DEALLOC_SCDRP

        .SBTTL  ALLOC_POOL      - Allocate a block of nonpaged pool
;++
; ALLOC_POOL
;
; This routine allocates a block of nonpaged pool no smaller than the
; size of a fork block (allowing COM$DRVDEALMEM to fork on this block
; during deallocation.) An extra quadword at the top of the block is
; reserved to save the size field, relieving the caller of this respons-
; ibility.  The caller is presented with the address just beyond the
; reserved quadword.  Although a word would be sufficient for this field,
; a quadword is used for  alignment purposes (some blocks are used as IRPs,
; which are placed on self-relative queues and require quadword alignment.)
;
; If an allocation failure occurs, the thread is stalled and wakes up once
; a second to retry the allocation.
;
; INPUTS:
;
;       R1      - Size of block to allocate
;       R3      - UCB address
;
; OUTPUTS:
;
;       R0      - Destroyed
;       R1      - Size of block allocated
;       R2      - Address of allocated block
;       -8(R2)  - Length of allocated block (used by DEALLOC_POOL)
;       All other registers preserved
;--

ALLOC_POOL:
```

```
        .ENABLE LSB                     ; ALLOC_POOL
        ADDL    #8,R1                   ; Reserve a quadword to save size
        CMPL    R1,#FKB$C_LENGTH        ; Requested size smaller than fork
                                        ; block?
        BGEQ    10$                     ; Branch if not
        MOVL    #FKB$C_LENGTH,R1        ; Use fork block size as minimum
10$:    PUSHL   R1                      ; Save allocation length
        PUSHL   R3                      ; Save UCB address
        JSB     G^EXE$ALONONPAGED       ; Allocate a block
        POPL    R3                      ; Restore register
        BLBC    R0,20$                  ; Branch if error
        ADDL    #4,SP                   ; Remove allocation length from stack
        PUSHR   #^M<R0,R1,R2,R3,R4,R5>  ; Save registers
        MOVC5   #0,.,#0,R1,(R2)         ; Initialize the packet
        POPR    #^M<R0,R1,R2,R3,R4,R5>  ; Restore registers
        MOVL    R1,(R2)+                ; Save size of block
        ADDL    #4,R2                   ; Skip a longword
        RSB                             ; Return to caller
;+
; A pool allocation failure occurred. Come back once a second and retry the
; operation until successful.
;-
20$:    SUBPUSH (SP)+               ; Save allocation length (PUSHL R1 above)
        SUBSAVE                     ; Save return address
        SK_WAIT #1,UCB=R3           ; Wait a second
        SUBPOP  -(SP)               ; Restore return address
        SUBPOP  R1                  ; Restore allocation length
        BRW     10$                 ; Try again
        .DISABLE LSB                ; ALLOC_POOL

        .SBTTL  DEALLOC_POOL    - Deallocate a block of nonpaged pool
;++
; DEALLOC_POOL
;
; This routine deallocates a block of nonpaged pool. The size of the block
; is stored in the reserved quadword at a negative offset from the
; beginning of the block.
;
; INPUTS:
;
;       R0      - Address of block to deallocate
;       -8(R0)  - Length of block to deallocate
;
; OUTPUTS:
;
;       R0      - Destroyed
;       All other registers preserved
;--

DEALLOC_POOL:

        .ENABLE LSB                     ; DEALLOC_POOL
        PUSHQ   R1                      ; Save R1,R2
        SUBL    #4,R0                   ; Skip a longword
        MOVL    -(R0),IRP$W_SIZE(R0)    ; Copy size field
        CLRB    IRP$B_TYPE(R0)          ; Clear type field (prevents block
                                        ; from being interpreted as shared
                                        ; memory during deallocation)
        JSB     G^EXE$DEANONPAGED       ; Deallocate the block
        POPQ    R1                      ; Restore R1,R2
        RSB
        .DISABLE LSB                    ; DEALLOC_POOL
```

```
        .SBTTL  SETUP_CMD       - Common setup for all SCSI commands
;++
; SETUP_CMD
;
; This routine performs common setup prior to the sending of a SCSI command.
; Setup includes allocating a command buffer, filling in the pointers in the
; SCDRP to the command and status fields, copying the SCSI command to the
; command buffer, allocating an S0 "user" buffer if the command requires
; transferring data to or from the class driver, filling in the SCDRP fields
; used to map this buffer, and mapping the buffer.
;
; Since this routine calls SPI$ALLOCATE_COMMAND_BUFFER, which can suspend
; the thread, the return PC must be saved in the SCDRP.
;
; INPUTS:
;
;       R2      - Pointer to entry in SCSI_CMD table
;       R4      - PDT address
;       R5      - SCDRP address
;
; OUTPUTS:
;
;       R0      - Status
;       R1,R2   - Destroyed
;
;       SCDRP$L_CMD_BUF - Address of SCSI command buffer
;       SCDRP$L_CMD_PTR - Address of SCSI command
;       SCDRP$L_STS_PTR - Address to save SCSI status byte
;       SCDRP$L_SVA_USER- Address of S0 "user" buffer
;       SCDRP$L_BCNT    - Length of S0 "user" buffer
;       SCDRP$W_BOFF    - Byte offset of S0 "user" buffer
;       SCDRP$L_SVAPTE  - SVAPTE of S0 "user" buffer
;       IRP$V_FUNC   - SET/CLEAR to indicate READ/WRITE from S0 "user" buffer
;       SCDRP$L_DMA_TIMEOUT - Time in seconds for a DMA timeout.
;       SCDRP$L_DISCON_TIMEOUT - Time in seconds for a disconnect to time out
;
;--
        .ENABLE LSB                         ; SETUP_CMD
SETUP_CMD:

SCSI_CMD_BUF_OVHD = 4 + 4                   ; 4 bytes to save status byte +
                                            ; 4 bytes for SCSI command length
        SUBSAVE                             ; Save return address
        MOVZBL  (R2),R1                     ; Get size SCSI command
        ADDL    #SCSI_CMD_BUF_OVHD,R1       ; Add in command buffer overhead
        SUBPUSH R2                          ; Save R2
        SPI$ALLOCATE_COMMAND_BUFFER         ; Allocate a command buffer
        MOVL    R2,R1                       ; Copy command buffer address
        SUBPOP  R2                          ; Restore R2
        MOVB    #^XFF,(R1)                  ; Initialize status field
        MOVAL   (R1)+,-                     ; Address to put SCSI status byte
                SCDRP$L_STS_PTR(R5)         ;
        MOVL    R1,SCDRP$L_CMD_PTR(R5)      ; Save address of SCSI command
        MOVZBL  (R2)+,R0                    ; Get SCSI command length
        MOVL    R0,(R1)+                    ; Save length in command buffer
        ASHL    #-1,R0,R0                   ; Change byte count to word count
10$:    MOVW    (R2)+,(R1)+                 ; Copy a byte of SCSI command
        SOBGTR  R0,10$                      ; Repeat for entire SCSI command
```

```
;+
; There is a dependency here that the format of the SCSI_COMMAND record
; does not change.
; Copy the per command timeout values from the SCSI_CMD block to the
; SCSI Class Driver Request Packet.
;
; R2 points at the direction field in the SCSI_CMD block.
;-
        MOVL    3(R2),-                 ; Time in seconds for a DMA timeout.
                 SCDRP$L_DMA_TIMEOUT(R5)
        MOVL    7(R2),-                 ; Disconnect timeout in seconds.
                 SCDRP$L_DISCON_TIMEOUT(R5)
;+
; Determine if a buffer has already been mapped. If no buffer has been
; mapped, then allocate a system buffer and map it to receive the data
; from the target device.
;-
        BBC     #SCDRP$V_BUFFER_MAPPED,-; If buffer is mapped then do
                SCDRP$L_SCSI_FLAGS(R5),-; special setup for this command
                20$

;+
; During the STARTIO operation in the class driver, the user's QIO
; parameters must be copied from the IRP to SCDRP (SCSI Class Driver
; Request Packet).  The user data is then mapped, the SCSI CMD packet is
; allocated, and the command is sent to a target, over the connection
; established during UNIT INIT.
;-
        MOVL    UCB$L_IRP(R3),R2        ; Get current I/O's IRP address
        CLRL    SCDRP$L_ABCNT(R5)       ; Initialize accumulated byte count
        MOVW    IRP$W_FUNC(R2),-        ; Copy function code and modifiers,
                SCDRP$W_FUNC(R5)        ; MEDIA, SVAPTE, and BOFF fields,
        MOVW    IRP$W_STS(R2),-         ; and STS
                SCDRP$W_STS(R5)         ;
        MOVL    IRP$L_MEDIA(R2),-       ; from the IRP to the SCDRP
                SCDRP$L_MEDIA(R5)       ;
        MOVL    IRP$L_SVAPTE(R2),-      ;
                SCDRP$L_SVAPTE(R5)      ;
        MOVW    IRP$W_BOFF(R2),-        ;
                SCDRP$W_BOFF(R5)        ;
        MOVL    IRP$L_BCNT(R2),-        ; Copy user's BCNT from IRP to SCDRP
                SCDRP$L_BCNT(R5)       ;
        CMPL    SCDRP$L_BCNT(R5),-      ; Transfer length greater than maximum
                UCB$L_MAXBCNT(R3)       ; supported?
        BGTR    300$                    ; GTR, therefore I/O must be segmented
        CLRL    SCDRP$L_PAD_BCNT(R5)    ; No padding of last page required
        ADDL3   #<4+4>,-     ; Address of transfer length field in
                SCDRP$L_CMD_PTR(R5),R1  ; SCSI command
        MOVB    SCDRP$L_BCNT(R5),(R1)   ; Copy user-supplied byte count to
                                        ; command.
        BRW     50$                     ; Setup finished.
```

```
20$:    CVTWL   (R2),R1             ; Get length of send data buffer
        BLSS    50$                 ; Branch if negative, no system buffer
                                    ; involved, leave SCDRP$L_BCNT unchanged
        BEQL    30$                 ; Branch if zero length, zero SCDRP$L_BCNT
        SUBPUSH R2                    ; Save R2
        BSBW    ALLOC_POOL            ; Allocate a buffer to receive response
        MOVL    R2,R1                 ; Copy buffer address
        SUBPOP  R2                    ; Restore R2
        MOVL    R1,SCDRP$L_SVA_USER(R5) ; Save address of allocated buffer
        MOVZWL  (R2)+,SCDRP$L_BCNT(R5)  ; Save length of transfer
        CLRL    SCDRP$L_PAD_BCNT(R5)    ; No padding required
        BICW3   #^C<^X1FF>,R1,-   ; And byte offset within page
                SCDRP$W_BOFF(R5)        ;
        INSV    (R2),#IRP$V_FUNC,#1,-  ; Set/clear FUNC bit to indicate READ/
                SCDRP$W_STS(R5)        ; WRITE function
        PUSHL   R3                     ; Save R3
        MOVL    SCDRP$L_SVA_USER(R5),R2 ; Get user buffer address
        JSB     G^MMG$SVAPTECHK        ; Get SVAPTE of allocated system buffer
        MOVL    R3,SCDRP$L_SVAPTE(R5)  ; Save SVAPTE in SCDRP
        POPL    R3                     ; Restore R3
        BISB    #SCDRP$M_S0BUF!-       ; This buffer is an S0 "user" buffer
                 SCDRP$M_BUFFER_MAPPED,- ; and it has been mapped
                SCDRP$L_SCSI_FLAGS(R5)   ;
        SPI$MAP_BUFFER                ; Map the "user" buffer for read access

50$:    MOVZWL  #SS$_NORMAL,R0        ; Set success status
52$:    SUBRETURN

30$:    CLRL    SCDRP$L_BCNT(R5)        ; No data being transferred
        BRB     50$                     ; Use common exit

300$:   MOVZWL  #SS$_IVBUFLEN,R0        ; Bad byte count
        BRB     52$
        .DISABLE        LSB             ; SETUP_CMD
                .SBTTL  CLEANUP_CMD    - Common cleanup for all SCSI commands
;++
; CLEANUP_CMD
;
; This routine performs common cleanup after the sending of a SCSI command,
; including unmapping the user buffer and deallocating the command buffer.
;
; INPUTS:
;
;       R4      - PDT address
;       R5      - SCDRP address
;
; OUTPUTS:
;
;       R2      - Destroyed
;       All other registers preserved
;--
CLEANUP_CMD:
```

```
        .ENABLE LSB                         ; CLEANUP_CMD
        PUSHR   #^M<R0,R1,R3>               ; Save registers
        BBCC    #SCDRP$V_BUFFER_MAPPED,-; Branch if no buffer has been mapped
                SCDRP$L_SCSI_FLAGS(R5),-;
                10$
        SPI$UNMAP_BUFFER                    ; Unmap the mapped buffer
10$:    BBCC    #SCDRP$V_S0BUF,-            ; Branch if this is not an S0 "user"
                SCDRP$L_SCSI_FLAGS(R5),-; buffer
                20$                         ;
        MOVL    SCDRP$L_SVA_USER(R5),R0 ; Get address of S0 user buffer
        CLRL    SCDRP$L_SVA_USER(R5)    ; Buffer no longer owned
        BSBW    DEALLOC_POOL                ; Deallocate the buffer
20$:    MOVL    SCDRP$L_CMD_BUF(R5),R0  ; Get address of command buffer
        SPI$DEALLOCATE_COMMAND_BUFFER   ; Deallocate the command buffer
30$:    POPR    #^M<R0,R1,R3>               ; Restore registers
        RSB
        .DISABLE LSB                        ; CLEANUP_CMD

        .SBTTL  LOG_ERROR       - Write an entry to the error log file
;++
; LOG_ERROR
;
; This routine writes an entry to the error log file. If the device is
; offline, no error is logged. This prevents the error log file from being
; filled up while the class driver does its periodic polling of devices
; that have been set offline. The assumption is that the initial error that
; caused the device to be placed offline has been logged and that
; subsequent error log entries would be redundant.
;
; INPUTS:
;
;       R5      - UCB address
;       R7      - Error type
;       R8      - VMS status code
;
; OUTPUTS:
;
;       All registers preserved
;--
LOG_ERROR:
        .ENABLE LSB                         ; LOG_ERROR
        PUSHR   #^M<R0,R2,R9,R10>           ; Save registers
        MOVB    UCB$B_DEVTYPE(R5),R9        ; Save SCSI device type
10$:    MOVB    UCB$B_DEVCLASS(R5),R10  ; Save DEVCLASS field
        MOVL    UCB$L_DDT(R5),R0            ; Get DDT address
        MOVW    #ERR_K_COMMAND_LENGTH,- ; Length of packet containing SCSI
                                            ; command
                DDT$W_ERRORBUF(R0)          ; in the DDT
        JSB     G^ERL$DEVICERR              ; Log a device error
        BBCC    #UCB$V_ERLOGIP,-            ; Clear error log in progress
                UCB$W_STS(R5),30$           ;
        MOVL    UCB$L_EMB(R5),R2            ; Get address of error message buffer
        BEQL    30$                         ; Branch if none available
        JSB     G^ERL$RELEASEMB             ; Release the error log buffer
30$:    POPR    #^M<R0,R2,R9,R10>           ; Restore registers
40$:    RSB                                 ; Return to caller
        .DISABLE LSB                        ; LOG_ERROR

        .SBTTL  SK_REG_DUMP     - Device register dump routine
```

```
;++
; SK_REG_DUMP
;
; This routine dumps device-specific information into an error log packet.
; The format of this information is as follows:
;
;       +-----------------------+
;       |       Longword count  | 4 bytes
;       +-----------------------+
;       |       Revision        | 1 byte
;       +-----------------------+
;       |       HW revision     | 4 bytes
;       +-----------------------+
;       |       Error Type      | 1 byte
;       +-----------------------+
;       |       SCSI ID         | 1 byte
;       +-----------------------+
;       |       SCSI LUN        | 1 byte
;       +-----------------------+
;       |       SCSI SUBLUN     | 1 byte
;       +-----------------------+
;       |       Port status     | 4 bytes
;       +-----------------------+
;       |     SCSI CMD LENGTH   | 1 byte
;       +-----------------------+
;       |     SCSI CMD BYTES    | Up to 12 bytes
;       +-----------------------+
;       |       SCSI STS        | 1 byte
;       +-----------------------+
;       |   Error Text Count    | 1 byte
;       +-----------------------+
;       |     Error Text        | Up to 60 bytes
;       +-----------------------+
;
; Inputs:
;
;       R0      - Output buffer address
;       R5      - UCB address
;
; Outputs:
;
;       R1-R3   - Destroyed
;       All other registers preserved
;--
```

```
SK_REG_DUMP:
        .ENABLE LSB             ; SK_REG_DUMP
        MOVL    #<<ERR_K_COMMAND_LENGTH/4>+1>,-
                (R0)+                   ; Length of error log packet in words
        MOVB    #0,(R0)+                ; Save revision level
        CLRL    (R0)+                   ; Save hardware revision level
        MOVB    R7,(R0)+                ; Save error type
        MOVZWL  UCB$W_UNIT(R5),R1       ; Get unit number
        CLRL    R2                      ; Prepare for extended divide
        EDIV    #100,R1,R1,R2           ; Extract SCSI bus ID from unit number
        MOVB    R1,(R0)+                ; Save SCSI bus ID
        MOVL    R2,R1                   ; Copy LUN, SUBLUN
        CLRL    R2                      ; Prepare for extended divide
        EDIV    #10,R1,R1,R2            ; Extract LUN and SUBLUN
        MOVB    R1,(R0)+                ; Save LUN field
        MOVB    R2,(R0)+                ; Save SUBLUN field
        MOVL    R8,(R0)+                ; Save port status code
        MOVL    UCB_L_SCDRP(R5),R1      ; Get active SCDRP address
        BEQL    50$                     ; Branch if none active
        MOVL    SCDRP$L_CMD_PTR(R1),R2  ; Get address of SCSI command
        BEQL    50$                     ; Branch if none active
        MOVL    (R2)+,R3                ; Get number of SCSI command bytes
        MOVB    R3,(R0)+                ; Save command length
10$:    MOVB    (R2)+,(R0)+             ; Save a command byte
        SOBGTR  R3,10$                  ; Continue for entire SCSI command
        MOVL    SCDRP$L_STS_PTR(R1),R2  ; Get address of status byte
        MOVB    (R2),(R0)+              ; Save SCSI status byte
        MOVB    (R11)+,R3               ; Get count of number of text bytes.
        BEQL    50$                     ; If no text finished
        MOVB    R3,(R0)+                ; Save text length
20$:    MOVB    (R11)+,(R0)+            ; Save a text byte in error packet
        SOBGTR  R3,20$                  ; Continue for entire text string
                                        ; command
50$:    RSB                             ; Return
        .DISABLE        LSB             ; SK_REG_DUMP
SK_PATCH:
        .BLKB   200                     ; Patch space

SK_END:                                 ; Last location in driver
        .END
```

C Sample Driver for the RL11, RL01, and RL02 Disk Drives

This example driver, DLDRIVER, drives a disk device on both the UNIBUS and the Q22 bus.

```
        .TITLE  DLDRIVER - VAX/VMS RL11/RL01,RL02 DISK DRIVER
        .IDENT  'X-7'
;
;*****************************************************************************
;*                                                                           *
;*  COPYRIGHT (c) 1978, 1980, 1982, 1984 BY                                  *
;*  DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASSACHUSETTS.                   *
;*  ALL RIGHTS RESERVED.                                                     *
;*                                                                           *
;*  THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED AND COPIED    *
;*  ONLY IN  ACCORDANCE WITH  THE  TERMS  OF  SUCH  LICENSE  AND WITH THE    *
;*  INCLUSION OF THE ABOVE COPYRIGHT NOTICE. THIS SOFTWARE OR  ANY  OTHER    *
;*  COPIES THEREOF MAY NOT BE PROVIDED OR OTHERWISE MADE AVAILABLE TO ANY    *
;*  OTHER PERSON.  NO TITLE TO AND OWNERSHIP OF  THE  SOFTWARE IS  HEREBY    *
;*  TRANSFERRED.                                                             *
;*                                                                           *
;*  THE INFORMATION IN THIS SOFTWARE IS  SUBJECT TO CHANGE WITHOUT NOTICE    *
;*  AND  SHOULD  NOT  BE  CONSTRUED AS  A COMMITMENT BY DIGITAL EQUIPMENT    *
;*  CORPORATION.                                                             *
;*                                                                           *
;*  DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE  OR  RELIABILITY OF ITS    *
;*  SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DIGITAL.                  *
;*                                                                           *
;*                                                                           *
;*****************************************************************************
;
;
; FACILITY:
;
;       VAX/VMS RL11/RL01,RL02 DISK DRIVER
;
;**
.PAGE
; ABSTRACT:
;
;       THIS MODULE CONTAINS THE TABLES AND ROUTINES NECESSARY TO
;       PERFORM ALL DEVICE-DEPENDENT PROCESSING OF AN I/O REQUEST
;       FOR RL11/RL01,RL02 DISK TYPES ON A VAX/VMS SYSTEM.
;
;       THE DISKS HAVE THE FOLLOWING PHYSICAL GEOMETRY:
;
;                       TRACKS/     SECTORS/      BYTES/      MAXIMUM
;          # CYL         CYLINDER    TRACK         SECTOR      BLOCKS
;
; RL01      256             2          40            256       10240
; RL02      512             2          40            256       20480
;
;       SINCE THE SECTOR SIZE IS ONLY 1/2 BLOCK, LOGICAL TO PHYSICAL
;       CONVERSION OF THE DISK ADDRESS IS DONE IN THE DRIVER STARTIO
;       ROUTINE RATHER THAN IN THE IOC$CVTLOGPHY FDT ROUTINE.
;
;       OVERLAPPED SEEKS ARE NOT ATTEMPTED BECAUSE THE DEVICE DOES
```

```
;       NOT INTERRUPT AT THE COMPLETION OF A SEEK.
;
;       ALSO, THE DEVICE DOES NOT PERFORM AN IMPLICIT SEEK WHEN PERFORMING
;       A READ OR WRITE FUNCTION,SO SEEK FUNCTIONS ARE ISSUED BY THIS
;       DRIVER WHERE NECESSARY PRIOR TO ISSUING A READ OR WRITE FUNCTION.
;       THE READ OR WRITE FUNCTION IS THEN ISSUED AS SOON AS THE RL11
;       CONTROLLER BECOMES READY (WHILE THE SEEK IS IN PROGRESS), AND A
;       WAIT FOR INTERRUPT (UPON COMPLETION OF THE READ OR WRITE) IS
;       ISSUED.  IF A SEEK FUNCTION IS REQUESTED SEPARATELY FROM A READ OR
;       WRITE, A DUMMY READ HEADER FUNCTION IS ISSUED FOLLOWING THE SEEK
;       FUNCTION AND A WAIT FOR INTERRUPT (UPON COMPLETION OF THE READ
;       HEADER) IS ISSUED.
;
;       THE IO$X_INHSEEK FUNCTION MODIFIER IS TREATED AS A NO-OP BY
;       THIS DRIVER, SINCE AN EXPLICIT SEEK IS NECESSARY FOR THE RL02
;       TO TRANSFER DATA PROPERLY.
;
;       THE RL'S DO NOT READ OR WRITE BEYOND THE END OF TRACK (THEY DO NOT
;       AUTOMATICALLY SEEK THE NEXT TRACK), SO  ALL READ AND WRITE FUNCTIONS
;       ARE BROKEN UP BY THIS DRIVER INTO PARTIAL TRANSFERS TO THE END OF
;       TRACK, FOLLOWED BY A SEEK TO THE NEXT TRACK, THEN ANOTHER READ OR
;       WRITE FUNCTION UNTIL THE TOTAL DATA TRANSFER IS COMPLETE.
;
;--
        .PAGE
        .SBTTL  EXTERNAL AND LOCAL DEFINITIONS

;
; EXTERNAL SYMBOLS
;

        $ADPDEF                         ;DEFINE ADAPTER CONTROL BLOCK
        $CRBDEF                         ;DEFINE CHANNEL REQUEST BLOCK
        $DCDEF                          ;DEFINE DEVICE CLASS
        $DDBDEF                         ;DEFINE DEVICE DATA BLOCK
        $DEVDEF                         ;DEFINE DEVICE CHARACTERISTICS
        $DPTDEF                         ;DEFINE DRIVER PROLOGUE TABLE
        $DYNDEF                         ;DEFINE DYNAMIC DATA STRUCTURE TYPES
        $EMBDEF                         ;DEFINE ERROR MESSAGE BUFFER
        $IDBDEF                         ;DEFINE INTERRUPT DATA BLOCK
        $IODEF                          ;DEFINE I/O FUNCTION CODES
        $IRPDEF                         ;DEFINE I/O REQUEST PACKET
        $PRDEF                          ;DEFINE PROCESSOR REGISTERS
        $PTEDEF                         ;DEFINE SYSTEM PTES
        $SSDEF                          ;DEFINE SYSTEM STATUS CODES
        $UCBDEF                         ;DEFINE UNIT CONTROL BLOCK
        $VADEF                          ;DEFINE VIRTUAL ADDRESS BITS
        $VECDEF                         ;DEFINE INTERRUPT VECTOR BLOCK


;
; LOCAL MACROS
;

;       EXFUNCL
;       BRANCH TO SUBROUTINE WHICH REQUESTS CHANNEL (IF NOT ALREADY OWNED),
;       EXECUTES FCODE (OR R3) FUNCTION, AND BRANCHES TO BDST ON ERROR

        .MACRO  EXFUNCL BDST,FCODE
                .IF NB  FCODE           ;IS FCODE NONBLANK?
                MOVZBL  #CD'FCODE,R3    ;IF NB - SPECIFY FCODE FUNCTION
                .ENDC                   ;IF B - SPECIFY FNTN IN EXISTING R3
                BSBW    FEXL            ;EXECUTE FUNCTION
                .BYTE   BDST-.-1        ;WHERE TO GO IF ERROR
        .ENDM
```

```
;       GENF
;       GENERATE FUNCTION TABLE ENTRY AND CASE TABLE INDEX SYMBOL

        .MACRO  GENF FCODE
                CD'FCODE=.-FTAB/2
                .WORD   FCODE!RL_CS_M_IE  ;FCODE WITH INT ENABLE BIT
        .ENDM

;       CKPWR
;       DISABLE INTERRUPTS, CHECK IF POWER HAS FAILED,
;       AND PUT DEVICE UNIT NUMBER IN R2<9:8>
;
        .MACRO  CKPWR SAVE_R0=YES,?L1
                CLRL    R2              ;CLEAR R2 FOR UNIT NUMBER
                INSV    UCB$W_UNIT(R5),-  ;PUT UNIT # IN R2<9:8>
                        #8,#2,R2        ;...
                DEVICELOCK -
                        LOCKADDR=UCB$L_DLCK(R5),- ; LOCK DEVICE ACCESS
                        LOCKIPL=UCB$B_DIPL(R5),- ; RAISE IPL
                        SAVIPL=-(SP),-  ;SAVE CURRENT IPL
                        PRESERVE='SAVE_R0
                SETIPL  #31,-           ;DISABLE ALL INTERRUPTS
                        ENVIRON=UNIPROCESSOR
                BBC     #UCB$V_POWER,-  ;IF CLR - NO POWER FAILURE
                        UCB$W_STS(R5),L1  ;...
                ; POWERFAILURE!
                DEVICEUNLOCK -
                        LOCKADDR=UCB$L_DLCK(R5),- ; UNLOCK DEVICE ACCESS
                        NEWIPL=(SP)+,-  ;RESTORE IPL
                        PRESERVE='SAVE_R0
                BRW     RETREG          ;EXIT
L1:                                     ;RETURN FOR NO POWER FAILURE
        .ENDM

;
; LOCAL SYMBOLS
;
RL_NUM_REGS     =4                      ;NUMBER OF DEVICE REGISTERS
RL_SLM          =5                      ;STATE=SEEK LINEAR MODE (READY TO GO)
UCB$B_DL_DCHEK  =UCB$W_OFFSET+1         ;REDEFINE FOR DATA CHECK USE

;
; UCB OFFSETS WHICH FOLLOW THE STANDARD UCB FIELDS
;
        $DEFINI UCB                     ;START OF UCB DEFINITIONS
```

Sample Driver for the RL11, RL01, and RL02 Disk Drives

```
.=UCB$K_LCL_DISK_LENGTH                 ;BEGIN DEFINITIONS AT END OF UCB
$DEF    UCB$W_DL_PBCR   .BLKW   1       ;PARTIAL BYTE COUNT
$DEF    UCB$W_DL_CS     .BLKW   1       ;CONTROL STATUS REGISTER
$DEF    UCB$W_DL_BA     .BLKW   1       ;BUS ADDRESS REGISTER
$DEF    UCB$W_DL_DA     .BLKW   1       ;DISK ADDRESS REGISTER
$DEF    UCB$W_DL_MP     .BLKW   1       ;MULTIPURPOSE REGISTER
$DEF    UCB$W_DL_DPN    .BLKW   1       ;DATA PATH NUMBER
$DEF    UCB$L_DL_SVAPTE                 ;SAVED SVAPTE OF THE USER'S BUFFER
$DEF    UCB$L_DL_DPR    .BLKL   1       ;DATAPATH REGISTER
$DEF    UCB$L_DL_BUFADR                 ;USER BUFFER ADDRESS
$DEF    UCB$L_DL_FMPR   .BLKL   1       ;FINAL MAP REGISTER
$DEF    UCB$A_DL_MOVRTN                 ;BUFFER MOVE ROUTINE ADDRESS
$DEF    UCB$L_DL_PMPR   .BLKL   1       ;PREVIOUS MAP REGISTER
$DEF    UCB$B_DL_DPPE   .BLKB   1       ;DATAPATH PURGE ERROR
$DEF    UCB$W_DL_DB     .BLKW   3       ;DATA BUFFER REGISTER
$DEF    UCB$B_DL_XBA    .BLKB   1       ;BUS ADDRESS EXTENSION BITS
$DEF    UCB$W_DL_SBA    .BLKW   1       ;SAVED BUFFER ADDRESS
$DEF    UCB$A_DL_BUF_VA .BLKL   1       ;PHYSICAL BUFFER VIRTUAL ADDRESS
$DEF    UCB$A_DL_BUF_PA .BLKL   1       ;PHYSICAL BUFFER PHYSICAL ADDRESS
$DEF    UCB$W_DL_FLAGS  .BLKW   1       ;FLAGS
        $VIELD  UCB,0,<-                ;START THE FLAG DEFINITIONS
                <DL_22BIT,,M>,-         ;22 BIT ADDRESSING
                <DL_MAPPING,,M>,-       ;ADAPTER MAPPING
                >                       ;END OF FLAG DEFINITIONS
$DEF    UCB$K_DL_LEN    .BLKW   1       ;LENGTH OF UCB
$EQU    UCB$K_DL_BUFSZ 20               ;BUFFER SIZE = 40 SECTORS *
                                        ;256 BYTES/SECTOR / 512 BYTES/PAGE
        $DEFEND UCB                     ;END OF UCB DEFINITIONS

;
; RL11/RL01 REGISTER OFFSETS FROM CSR ADDRESS
;
        $DEFINI RL                      ; START OF REGISTER DEFINITIONS

$DEF    RL_CS           .BLKW   1       ;CONTROL STATUS REGISTER (CSR)
        _VIELD  RL_CS,0,<-              ;START OF CSR BIT DEFINITIONS
                <DRDY,,M>,-             ; DRIVE READY
                <FCODE,3>,-             ; FUNCTION CODE
                <XBA,2>,-               ; BUS ADDRESS EXTENSION BITS
                <IE,,M>,-               ; INTERRUPT ENABLE
                <CRDY,,M>,-             ; CONTROLLER READY
                <DS,2>,-                ; DRIVE SELECT
                <OPI,,M>,-              ; OPERATION INCOMPLETE
                <CRC,,M>,-              ; DATA CRC OR HEADER CRC
                <DLT,,M>,-              ; DATA LATE OR HEADER NOT FOUND
                <NXM,,M>,-              ; NONEXISTENT MEMORY
                <DE,,M>,-               ; DRIVE ERROR
                <CE,,M>-                ; COMPOSITE ERROR
        >                               ;END CSR BIT DEFINITIONS

$DEF    RL_BA           .BLKW   1       ;BUS ADDRESS REGISTER (BAR)

$DEF    RL_DA           .BLKW   1       ;DISK ADDRESS REGISTER (DAR)
        _VIELD  RL_DA,0,<-              ;START OF DAR BIT DEFINITIONS
                <MRK,,M>,-              ; MARK (ALWAYS 1)
                <STS,,M>,-              ; GET STATUS
                <,1>,-                  ; RESERVED BIT
                <RST,,M>,-              ; RESET
                <,12>,-                 ; RESERVED BITS
        >                               ;END OF DAR BIT DEFINITIONS
```

```
$DEF    RL_MP           .BLKW   1       ;MULTIPURPOSE REGISTER (MPR)
        _VIELD  RL_MP,0,<-              ;START OF MPR BIT DEFINITIONS
                <STA,3>,-               ; DRIVE STATE
                <BH,,M>,-               ; BRUSH HOME
                <HO,,M>,-               ; HEADS OUT
                <CO,,M>,-               ; COVER OPEN
                <HS,,M>,-               ; HEAD SELECT
                <TYP,,M>,-              ; DRIVE TYPE
                <DSE,,M>,-              ; DRIVE SELECT ERROR
                <VC,,M>,-               ; VOLUME CHECK
                <WGE,,M>,-              ; WRITE GATE ERROR
                <SPE,,M>,-              ; SPIN ERROR
                <SKTO,,M>,-             ; SEEK TIME OUT
                <WL,,M>,-               ; WRITE LOCK
                <CHE,,M>,-              ; CURRENT HEAD ERROR
                <WDE,,M>-               ; WRITE DATA ERROR
        >                               ;END MPR BIT DEFINITIONS

$DEF    RL_BAE          .BLKW   1       ; BUS ADDRESS EXTENSION REGISTER(BAE)
        $DEFEND RL                      ;END RL11/RL01 REGISTER DEFINITIONS

;
; HARDWARE FUNCTION CODES
;
F_NOP=0*2                               ;NO OPERATION
F_UNLOAD=F_NOP                          ;NO OPERATION
F_SEEK=3*2                              ;SEEK CYLINDER
F_RECAL=F_NOP                           ;NO OPERATION
F_DRVCLR=2*2                            ;DRIVE CLEAR (GET STATUS)
F_RELEASE=F_NOP                         ;NO OPERATION
F_OFFSET=F_NOP                          ;NO OPERATION
F_RETCENTER=F_NOP                       ;NO OPERATION
F_PACKACK=2*2                           ;PACK ACKNOWLEDGE (SET VOLUME VALID)
F_SEARCH=F_NOP                          ;NO OPERATION
F_WRITECHECK=1*2                        ;WRITE CHECK
F_WRITEDATA=5*2                         ;WRITE DATA
F_WRITEHEAD=F_NOP                       ;NO OPERATION
F_READDATA=6*2                          ;READ DATA
F_READHEAD=4*2                          ;READ HEADER
F_AVAILABLE=F_NOP                       ;NO OPERATION
F_GETSTATUS=2*2                         ;GET STATUS (DRIVER INTERNAL USE)

        .PAGE
        .SBTTL  STANDARD TABLES

;
; DRIVER PROLOGUE TABLE
;
;       THE DPT DESCRIBES DRIVER PARAMETERS AND I/O DATABASE FIELDS
;       THAT ARE TO BE INITIALIZED DURING DRIVER LOADING AND RELOADING
;
        DPTAB   -                       ;DPT CREATION MACRO
                END=DL_END,-            ;END OF DRIVER LABEL
                ADAPTER=UBA,-           ;ADAPTER TYPE = UNIBUS
                FLAGS=DPT$M_SVP,-       ;SYSTEM PAGE-TABLE ENTRY REQUIRED
                UCBSIZE=UCB$K_DL_LEN,-  ;LENGTH OF UCB
                NAME=DLDRIVER           ;DRIVER NAME
```

```
        DPT_STORE INIT                  ;START CONTROL BLOCK INIT VALUES
        DPT_STORE DDB,DDB$L_ACPD,L,<^A\F11\>  ;DEFAULT ACP NAME
        DPT_STORE DDB,DDB$L_ACPD+3,B,DDB$K_CART ;ACP CLASS
        DPT_STORE UCB,UCB$B_FLCK,B,SPL$C_IOLOCK8 ;FORK LOCK INDEX
        DPT_STORE UCB,UCB$L_DEVCHAR,L,- ;DEVICE CHARACTERISTICS
                <DEV$M_FOD-             ; FILES ORIENTED
                !DEV$M_DIR-             ; DIRECTORY STRUCTURED
                !DEV$M_AVL-             ; AVAILABLE
                !DEV$M_ELG-             ; ERROR LOGGING
                !DEV$M_SHR-             ; SHAREABLE
                !DEV$M_IDV-             ; INPUT DEVICE
                !DEV$M_ODV-             ; OUTPUT DEVICE
                !DEV$M_RND>             ; RANDOM ACCESS
        DPT_STORE UCB,UCB$L_DEVCHAR2,L,-; DEVICE CHARACTERISTICS
                <DEV$M_NNM>             ; PREFIX NAME WITH "node$"
        DPT_STORE UCB,UCB$B_DEVCLASS,B,DC$_DISK  ;DEVICE CLASS
        DPT_STORE UCB,UCB$W_DEVBUFSIZ,W,512  ;DEFAULT BUFFER SIZE
        DPT_STORE UCB,UCB$B_SECTORS,B,40  ;NUMBER OF SECTORS PER TRACK
        DPT_STORE UCB,UCB$B_TRACKS,B,2  ;NUMBER OF TRACKS PER CYLINDER
        DPT_STORE UCB,UCB$B_DIPL,B,21   ;DEVICE IPL
        DPT_STORE UCB,UCB$B_ERTMAX,B,8  ;MAX ERROR RETRY COUNT
        DPT_STORE UCB,UCB$W_DEVSTS,W,-  ;INHIBIT LOG TO PHYS CONVERSION IN FDT
                <UCB$M_NOCNVRT>         ;...

        DPT_STORE REINIT                ;START CONTROL BLOCK RE-INIT VALUES
        DPT_STORE CRB,CRB$L_INTD+4,D,DL_INT  ;INTERRUPT SERVICE ROUTINE ADDRESS
        DPT_STORE CRB,CRB$L_INTD+VEC$L_INITIAL,-  ;CONTROLLER INIT ADDRESS
                    D,DL_RL11_INIT      ;...
        DPT_STORE CRB,CRB$L_INTD+VEC$L_UNITINIT,- ;UNIT INIT ADDRESS
                    D,DL_RL0X_INIT      ;...
        DPT_STORE DDB,DDB$L_DDT,D,DL$DDT        ;DDT ADDRESS

        DPT_STORE END                   ;END OF INITIALIZATION TABLE

;
; DRIVER DISPATCH TABLE
;
;       THE DDT LISTS ENTRY POINTS FOR DRIVER SUBROUTINES WHICH ARE
;       CALLED BY THE OPERATING SYSTEM.
;

        DDTAB   -                       ;DDT CREATION MACRO
                DEVNAM=DL,-             ;NAME OF DEVICE
                START=DL_STARTIO,-      ;START I/O ROUTINE
                UNSOLIC=DL_UNSOLNT,-    ;UNSOLICITED INTERRUPT
                FUNCTB=DL_FUNCTABLE,-   ;FUNCTION DECISION TABLE
                CANCEL=0,-              ;CANCEL=NO-OP FOR FILES DEVICE
                REGDMP=DL_REGDUMP,-     ;REGISTER DUMP ROUTINE
                DIAGBF=<<RL_NUM_REGS+5+5+3+1>*4>,-  ;BYTES IN DIAG BUFFER
                ERLGBF=<<<RL_NUM_REGS+5+1>*4>+EMB$L_DV_REGSAV>  ;BYTES IN
                                                    ;ERROR LOG BUFFER

; DIAGNOSTIC BUFFER SIZE = <<4 RL02 REGISTER LONGWORDS + 5 UCB FIELD LONGWORDS
;                          + 5 IOC$DIAGBUFILL LONGWORDS + 3 BUFFER ALLOCATION
;                          LONGWORDS + 1 LONGWORD FOR # REGISTERS IN DL_REGDUMP>
;                          * 4 BYTES/LONGWORD>
;
; ERROR LOG BUFFER SIZE  = <<<4 RL02 REGISTER LONGWORDS + 5 UCB FIELD LONGWORDS
;                          + 1 LONGWORD FOR # REGISTERS IN DL_REGDUMP>
;                          * 4 BYTES/LONGWORD> + BYTES NEEDED FOR ERROR LOGGER
;                          TO SAVE SOFTWARE REGISTERS>
```

```
;
; HARDWARE FUNCTION CODE TABLE
;
;       THIS TABLE MERGES THE FUNCTION CODE BITS WITH THE
;       INTERRUPT ENABLE BIT AND GENERATES THE CASE TABLE
;       INDEX SYMBOL.

FTAB:   GENF    F_NOP                   ;NO-OP
        GENF    F_UNLOAD                ;UNLOAD VOLUME (NOP)
        GENF    F_SEEK                  ;SEEK
        GENF    F_RECAL                 ;RECALIBRATE (NOP)
        GENF    F_DRVCLR                ;DRIVE CLEAR (RESET & GET STATUS)
        GENF    F_RELEASE               ;RELEASE PORT (NOP)
        GENF    F_OFFSET                ;OFFSET HEADS (NOP)
        GENF    F_RETCENTER             ;RETURN HEADS TO CENTERLINE (NOP)
        GENF    F_PACKACK               ;PACK ACKNOWLEDGE (RESET & GET STATUS)
        GENF    F_SEARCH                ;SEARCH (NOP)
        GENF    F_WRITECHECK            ;WRITE CHECK
        GENF    F_WRITEDATA             ;WRITE DATA
        GENF    F_READDATA              ;READ DATA
        GENF    F_WRITEHEAD             ;WRITE HEADERS (NOP)
        GENF    F_READHEAD              ;READ HEADERS
        GENF    F_NOP                   ;place holder
        GENF    F_NOP                   ;place holder
        GENF    F_AVAILABLE             ;AVAILABLE

        .PAGE
;
; FUNCTION DECISION TABLE
;
;       THE FDT LISTS VALID FUNCTION CODES, SPECIFIES WHICH
;       CODES ARE BUFFERED, AND DESIGNATES SUBROUTINES TO
;       PERFORM PREPROCESSING FOR PARTICULAR FUNCTIONS.
;
DL_FUNCTABLE:
        FUNCTAB ,-                      ;LIST LEGAL FUNCTIONS
                <NOP,-                  ; NO-OP
                UNLOAD,-                ; UNLOAD
                SEEK,-                  ; SEEK
                DRVCLR,-                ; DRIVE CLEAR
                PACKACK,-               ; PACK ACKNOWLEDGE
                SENSECHAR,-             ; SENSE CHARACTERISTICS
                SETCHAR,-               ; SET CHARACTERISTICS
                SENSEMODE,-             ; SENSE MODE
                SETMODE,-               ; SET MODE
                WRITECHECK,-            ; WRITE CHECK
                READHEAD,-              ; READ HEADER
                READLBLK,-              ; READ LOGICAL BLOCK
                WRITELBLK,-             ; WRITE LOGICAL BLOCK
                READPBLK,-              ; READ PHYSICAL BLOCK
                WRITEPBLK,-             ; WRITE PHYSICAL BLOCK
                READVBLK,-              ; READ VIRTUAL BLOCK
                WRITEVBLK,-             ; WRITE VIRTUAL BLOCK
                AVAILABLE,-             ; AVAILABLE
                ACCESS,-                ; ACCESS FILE / FIND DIRECTORY ENTRY
                ACPCONTROL,-            ; ACP CONTROL FUNCTION
                CREATE,-                ; CREATE FILE AND/OR DIRECTORY ENTRY
                DEACCESS,-              ; DEACCESS FILE
                DELETE,-                ; DELETE FILE AND/OR DIRECTORY ENTRY
                MODIFY,-                ; MODIFY FILE ATTRIBUTES
                MOUNT-                  ; MOUNT VOLUME
                >
        FUNCTAB ,-                      ;BUFFERED FUNCTIONS
                <NOP,-                  ; NO-OP
```

```
        UNLOAD,-                ; UNLOAD
        SEEK,-                  ; SEEK
        DRVCLR,-                ; DRIVE CLEAR
        PACKACK,-               ; PACK ACKNOWLEDGE
        SENSECHAR,-             ; SENSE CHARACTERISTICS
        SETCHAR,-               ; SET CHARACTERISTICS
        SENSEMODE,-             ; SENSE MODE
        SETMODE,-               ; SET MODE
        AVAILABLE,-             ; AVAILABLE
        ACCESS,-                ; ACCESS FILE / FIND DIRECTORY ENTRY
        ACPCONTROL,-            ; ACP CONTROL FUNCTION
        CREATE,-                ; CREATE FILE AND/OR DIRECTORY ENTRY
        DEACCESS,-              ; DEACCESS FILE
        DELETE,-                ; DELETE FILE AND/OR DIRECTORY ENTRY
        MODIFY,-                ; MODIFY FILE ATTRIBUTES
        MOUNT-                  ; MOUNT VOLUME
        >
FUNCTAB DL_ALIGN,-              ;TEST ALIGNMENT FUNCTIONS
        <READHEAD,-             ; READ HEADER
        READLBLK,-              ; READ LOGICAL BLOCK
        READPBLK,-              ; READ PHYSICAL BLOCK
        READVBLK,-              ; READ VIRTUAL BLOCK
        WRITECHECK,-            ; WRITE CHECK
        WRITELBLK,-             ; WRITE LOGICAL BLOCK
        WRITEPBLK,-             ; WRITE PHYSICAL BLOCK
        WRITEVBLK-              ; WRITE VIRTUAL BLOCK
        >
FUNCTAB +ACP$READBLK,-          ;READ FUNCTIONS
        <READHEAD,-             ; READ HEADER
        READLBLK,-              ; READ LOGICAL BLOCK
        READPBLK,-              ; READ PHYSICAL BLOCK
        READVBLK-               ; READ VIRTUAL BLOCK
        >
FUNCTAB +ACP$WRITEBLK,-         ;WRITE FUNCTIONS
        <WRITECHECK,-           ; WRITE CHECK
        WRITELBLK,-             ; WRITE LOGICAL BLOCK
        WRITEPBLK,-             ; WRITE PHYSICAL BLOCK
        WRITEVBLK-              ; WRITE VIRTUAL BLOCK
        >
FUNCTAB +ACP$ACCESS,-           ;ACCESS FUNCTIONS
        <ACCESS,-               ; ACCESS FILE / FIND DIRECTORY ENTRY
        CREATE-                 ; CREATE FILE AND/OR DIRECTORY ENTRY
        >
FUNCTAB +ACP$DEACCESS,-         ;DEACCESS FUNCTION
        <DEACCESS-              ; DEACCESS FILE
        >
FUNCTAB +ACP$MODIFY,-           ;MODIFY FUNCTIONS
        <ACPCONTROL,-           ; ACP CONTROL FUNCTION
        DELETE,-                ; DELETE FILE AND/OR DIRECTORY ENTRY
        MODIFY-                 ; MODIFY FILE ATTRIBUTES
        >
FUNCTAB +ACP$MOUNT,-            ;MOUNT FUNCTION
        <MOUNT-                 ; MOUNT VOLUME
        >
FUNCTAB +EXE$LCLDSKVALID,-      ;LOCAL DISK VALID FUNCTIONS
        <UNLOAD,-               ;UNLOAD VOLUME
         AVAILABLE,-            ;UNIT AVAILABLE
         PACKACK-               ;PACK ACKNOWLEDGE
        >
FUNCTAB +EXE$ZEROPARM,-         ;ZERO PARAMETER FUNCTIONS
        <NOP,-                  ; NO-OP
        UNLOAD,-                ; UNLOAD
        DRVCLR,-                ; DRIVE CLEAR
```

```
                PACKACK,-               ; PACK ACKNOWLEDGE
                AVAILABLE,-             ; AVAILABLE
                >
        FUNCTAB +EXE$ONEPARM,-          ;ONE PARAMETER FUNCTION
                <SEEK-                  ; SEEK
                >
        FUNCTAB +EXE$SENSEMODE,-        ;SENSE FUNCTIONS
                <SENSECHAR,-            ; SENSE CHARACTERISTICS
                SENSEMODE-              ; SENSE MODE
                >
        FUNCTAB +EXE$SETCHAR,-          ;SET FUNCTIONS
                <SETCHAR,-              ; SET CHARACTERISTICS
                SETMODE-                ; SET MODE
                >
        .PAGE

        .SBTTL  CONTROLLER INITIALIZATION ROUTINE
;++
;
; FUNCTIONAL DESCRIPTION:
;
;       THIS ROUTINE IS A NO-OP FOR THE RL11 BUT MUST BE INCLUDED
;       SINCE IT IS CALLED WHEN THE RL02 IS BOOTED AS A SYSTEM DEVICE.
;
;       THE OPERATING SYSTEM CALLS THIS ROUTINE:
;               - AT SYSTEM STARTUP
;               - DURING DRIVER LOADING
;               - DURING RECOVERY FROM POWER FAILURE
;
; INPUTS:
;
;       R4      - CSR ADDRESS (DEVICE CONTROL STATUS REGISTER)
;       R5      - IDB ADDRESS (INTERRUPT DATA BLOCK)
;       R6      - DDB ADDRESS (DEVICE DATA BLOCK)
;       R8      - CRB ADDRESS (CHANNEL REQUEST BLOCK)
;       ALL INTERRUPTS ARE LOCKED OUT
;
; OUTPUTS:
;
;       ALL REGISTERS EXCEPT R0-R3 ARE PRESERVED.
;       CONTROL IS RETURNED TO THE CALLER.
;
;--
DL_RL11_INIT:                           ;CONTROLLER INITIALIZATION
        ;
        ; FOR MICROVAX I, ALLOCATE A PHYSICALLY CONTIGUOUS BUFFER
        ; AREA FOR PERFORMING I/O.
        ;
        ADPDISP SELECT=ADAP_MAPPING,-   ; Allocate a physically contiguous
                ADDRLIST=<<YES,20$>>,-  ; buffer for those adapters that
                CRBADDR=R8,-            ; don't support mapping.
                SCRATCH=R0

10$:    MOVZWL  #UCB$K_DL_BUFSZ,R1      ;LOAD SIZE OF BUFFER
        JSB     G^EXE$ALOPHYCNTG        ;ALLOCATE PHYSICALLY CONTIGUOUS MEMORY
        BLBC    R0,20$                  ;EXIT ON ERROR
        MOVL    R2,CRB$L_AUXSTRUC(R8)   ;GET BUFFER VIRTUAL ADDRESS
        RSB                             ;RETURN TO CALLER

20$:    CLRL    CRB$L_AUXSTRUC(R8)      ;INDICATE MEMORY ALLOCATION FAILURE
        RSB                             ;RETURN TO CALLER
        .PAGE
        .SBTTL  UNIT INITIALIZATION ROUTINE
```

Sample Driver for the RL11, RL01, and RL02 Disk Drives

```
;++
;
; DL_RL0X_INIT - UNIT INITIALIZATION ROUTINE
;
; FUNCTIONAL DESCRIPTION:
;
;       THIS ROUTINE READIES THE RL01/RL02 UNITS FOR I/O OPERATIONS.
;
;       THE OPERATING SYSTEM CALLS THIS ROUTINE:
;               - AT SYSTEM STARTUP
;               - DURING DRIVER LOADING
;               - DURING RECOVERY FROM POWER FAILURE
;
; INPUTS:
;
;       R4      - CSR ADDRESS (CONTROLLER STATUS REGISTER)
;       R5      - UCB ADDRESS (UNIT CONTROL BLOCK)
;
; OUTPUTS:
;
;       THE DRIVE UNIT IS RESET, UCB FIELDS ARE INITIALIZED, AND THE
;       ROUTINE WAITS FOR ONLINE UNITS TO SPIN UP.  ALL REGISTERS
;       EXCEPT R0-R3 ARE PRESERVED.
;
;--
DL_RL0X_INIT:                           ;RL01/RL02 UNIT INITIALIZATION
        MOVW    #1@UCB$V_DL_MAPPING,-   ; DEFAULT TO ADAPTER MAPPING
                UCB$W_DL_FLAGS(R5)      ; AND 18 BIT ADDRESSING
        ADPDISP SELECT=ADAP_MAPPING,-
                ADDRLIST=<<YES,2$>>,-
                UCBADDR=R5,-
                SCRATCH=R0
        CLRW    UCB$W_DL_FLAGS(R5)      ; Clear adapter mapping bit
2$:     ADPDISP SELECT=ADDR_BITS,-
                ADDRLIST=<<18,3$>>,-
                ADPADDR=R0
        BISW    #1@UCB$V_DL_22BIT,-     ; FOR MICROVAX II 22-BIT
                UCB$W_DL_FLAGS(R5)      ; ADDRESSING AS WELL AS ADAPTER MAPPING
3$:
10$:    MOVZWL  UCB$W_STS(R5),R3        ;SAVE CURRENT UNIT STATUS
        BICW    #UCB$M_ONLINE!UCB$M_VALID,-  ;ASSUME OFFLINE/INVALID
                UCB$W_STS(R5)           ;...

;
; WAIT FOR CONTROLLER (6 SECONDS MAX) IF CHANNEL IS BUSY WITH ANOTHER UNIT
;
        MOVL    UCB$L_CRB(R5),R0        ;GET CRB ADDRESS
        BBC     #CRB$V_BSY,CRB$B_MASK(R0),20$  ;IF CLEAR - CHANNEL NOT BUSY
        TIMEDWAIT TIME=#600*1000,-      ;6 SECOND WAIT LOOP
                INS1=<TSTB      RL_CS(R4)>,- ;IS CONTROLLER READY
                INS2=<BLSS      15$>,-  ;IF LSS - YES
                DONELBL=15$             ;LABEL TO EXIT WAIT LOOP
        BLBC    R0,25$                  ;TIME EXPIRED - EXIT
;
; GET CURRENT DRIVE STATUS AND RESET DRIVE
;
```

```
20$:    MOVW    #RL_DA_M_RST!-          ;PUT RESET AND GET STATUS IN DAR
                RL_DA_M_STS!RL_DA_M_MRK,RL_DA(R4)  ;...
        CLRL    R1                      ;CLEAR R1 FOR UNIT NUMBER
        INSV    UCB$W_UNIT(R5),#8,#8,R1 ;GET UNIT NUMBER
        BISW3   R1,#F_GETSTATUS,RL_CS(R4)  ;EXECUTE GET STATUS FUNCTION
        BSBW    DL_WAIT                 ;WAIT FOR CONTROLLER
        TSTB    RL_CS(R4)               ;WAS CONTROLLER READY?
        BGEQ    25$                     ;IF GEQ - NO

;
; CLASSIFY DRIVE TYPE
;
        MOVL    #^X2324C001,-
                 UCB$L_MEDIA_ID(R5)     ;SET MEDIA IDENT "DL RL01"
        BITW    #RL_MP_M_TYP,RL_MP(R4)  ;IS DRIVE TYPE = RL02?
        BNEQ    30$                     ;IF NEQ - YES
        MOVB    S^#DT$_RL01,-
                 UCB$B_DEVTYPE(R5)      ;SET RL01 DEVICE TYPE
        MOVW    #256,UCB$W_CYLINDERS(R5);SET NUMBER OF RL01 CYLINDERS
        MOVZWL  #10240,UCB$L_MAXBLOCK(R5) ;SET MAX RL01 BLOCK NUMBER
        BRB     40$

25$:    BRB     70$                     ;BRANCH TO COMMON EXIT

30$:    MOVB    S^#DT$_RL02,-
                 UCB$B_DEVTYPE(R5)      ;SET RL02 DEVICE TYPE
        MOVW    #512,UCB$W_CYLINDERS(R5);SET NUMBER OF RL02 CYLINDERS
        MOVZWL  #20480,UCB$L_MAXBLOCK(R5) ;SET MAX RL02 BLOCK NUMBER
        INCL    UCB$L_MEDIA_ID(R5)      ;SET MEDIA IDENT "DL RL02"
40$:    BBC     #UCB$V_VALID,R3,60$     ; Branch around wait for drive to spin up
                                        ;  if the drive did NOT have a VALID
                                        ;  volume on it before POWER failure.

;
; INITIALIZE UCB FIELDS AND WAIT FOR ONLINE UNITS TO SPIN UP
;
45$:    BITW    #RL_CS_M_DRDY,RL_CS(R4) ; Is drive ready?
        BNEQ    50$                     ;IF NEQ - YES
        JSB     G^EXE$PWRTIMCHK         ;IS MAX TIME EXCEEDED?
        BLBS    R0,45$                  ;IF LBS - NO, STILL MORE TIME NEEDED
        BRB     60$                     ;POWER UP TIME EXCEEDED
50$:
        BISW    #UCB$M_VALID,UCB$W_STS(R5) ;SET UCB STATUS VOLUME VALID
```

```
60$:    BBS     #UCB$V_DL_MAPPING,-     ;ADAPTER MAPPING?
                UCB$W_DL_FLAGS(R5),65$  ;IF BS YES
        MOVL    UCB$L_CRB(R5),R1        ;GET CRB ADDRESS
        MOVL    CRB$L_AUXSTRUC(R1),R2   ;MEMORY ALLOC FAILURE DURING CTL INIT?
        BEQL    70$                     ;IF EQL YES, LEAVE OFFLINE
        MOVL    R2,UCB$A_DL_BUF_VA(R5)  ;SAVE BUFFER'S VIRTUAL ADDRESS
        EXTZV   #VA$V_VPN,#VA$S_VPN,R2,R1;GET VIRTUAL PAGE NUMBER OF BUFFER
        MOVL    G^MMG$GL_SPTBASE,R0     ;GET BASE ADDRESS OF SPTS
        MOVL    (R0)[R1],R0             ;GET THE PTE CONTENTS
        BICL3   #^C<VA$M_BYTE>,R2,R1    ;GET BUFFER OFFSET (BA00-BA08)
        ASSUME  PTE$S_PFN GE 13
        INSV    R0,#9,#13,R1            ;COPY BA09-BA21
        MOVL    R1,UCB$A_DL_BUF_PA(R5)  ;SAVE PHYSICAL ADDRESS OF BUFFER
65$:    BISW    #UCB$M_ONLINE,UCB$W_STS(R5)  ;SET UCB STATUS VOLUME VALID
70$:    RSB
        .PAGE
        .SBTTL  DRIVER SPECIFIC SUBROUTINES
;
; DL_WAIT - WAIT FOR CONTROLLER READY
;
; INPUTS:
;       R4      - DEVICE CSR ADDRESS
;
; FUNCTIONAL DESCRIPTION:
;
;       THIS ROUTINE IS CALLED FROM THE DRIVER UNIT INITIALIZATION ROUTINE
;       TO WAIT UNTIL THE RL11 CONTROLLER IS READY.  TO PREVENT HANGING UP
;       AT HIGH IPL, A MAXIMUM OF 30 USEC ELAPSES BEFORE CONTROL IS
;       RETURNED TO THE CALLER.
;
DL_WAIT:                                ;WAIT FOR CONTROLLER READY
        MOVQ    R0,-(SP)                ;SAVE R0, R1
        TIMEWAIT        #3,#RL_CS_M_CRDY,RL_CS(R4),W
        MOVQ    (SP)+,R0                ;RESTORE R0, R1
        RSB                             ;RETURN TO UNIT INIT OR STARTIO
        .PAGE
        .SBTTL  FDT ROUTINE - TEST TRANSFER BYTE COUNT ALIGNMENT

;++
;
; DL_ALIGN - FDT ROUTINE TO TEST XFER BYTE COUNT
;
; FUNCTIONAL DESCRIPTION:
;
;       THIS ROUTINE IS CALLED FROM THE FUNCTION DECISION TABLE DISPATCHER
;       TO CHECK THE BYTE COUNT PARAMETER SPECIFIED BY THE USER PROCESS
;       FOR AN EVEN NUMBER OF BYTES (WORD BOUNDARY).
;
; INPUTS:
;
;       R3      - IRP ADDRESS (I/O REQUEST PACKET)
;       R4      - PCB ADDRESS (PROCESS CONTROL BLOCK)
;       R5      - UCB ADDRESS (UNIT CONTROL BLOCK)
;       R6      - CCB ADDRESS (CHANNEL CONTROL BLOCK)
;       R7      - BIT NUMBER OF THE I/O FUNCTION CODE
;       R8      - ADDRESS OF FDT TABLE ENTRY FOR THIS ROUTINE
;       4(AP)   - ADDRESS OF FIRST FUNCTION DEPENDENT QIO PARAMETER
;
; OUTPUTS:
;
;       IF THE QIO BYTE COUNT PARAMETER IS ODD, THE I/O OPERATION IS
;       TERMINATED WITH AN ERROR. IF IT IS EVEN, CONTROL IS RETURNED
;       TO THE FDT DISPATCHER.
```

```
;
;--
DL_ALIGN:                                   ;CHECK BYTE COUNT AT P1(AP)
        BLBS    4(AP),10$                   ;IF LBS - ODD BYTE COUNT
        RSB                                 ;EVEN - RETURN TO CALLER
10$:    MOVZWL  #SS$_IVBUFLEN,R0            ;SET BUFFER ALIGNMENT STATUS
        JMP     G^EXE$ABORTIO               ;ABORT I/O
        .PAGE
        .SBTTL  START I/O ROUTINE

;++
;
; DL_STARTIO - START I/O ROUTINE
;
; FUNCTIONAL DESCRIPTION:
;
;       THIS FORK PROCESS IS ENTERED FROM THE EXECUTIVE AFTER AN I/O REQUEST
;       PACKET HAS BEEN DEQUEUED, AND PERFORMS THE FOLLOWING:
;
;               - ACTIVATES THE DISK AFTER SETTING UCB FIELDS, OBTAINING
;                 UBA AND CONTROLLER RESOURCES, AND SETTING RL11 REGISTERS
;
;               - WAITS FOR AN INTERRUPT
;
;               - REGAINS CONTROL AFTER THE ISR SERVICES THE INTERRUPT, AND
;                       -  REACTIVATES THE DISK IF THE ORIGINAL FUNCTION
;                          IS NOT YET COMPLETE, OR
;                       -  COMPLETES THE I/O REQUEST BY RELEASING RESOURCES,
;                          SETTING STATUS CODES, AND RETURNING TO THE EXECUTIVE.
;
; INPUTS:
;
;       R3              - IRP ADDRESS (I/O REQUEST PACKET)
;       R5              - UCB ADDRESS (UNIT CONTROL BLOCK)
;       IRP$L_MEDIA     - PARAMETER LONGWORD (LOGICAL BLOCK NUMBER)
;
; OUTPUTS:
;
;       R0      - FIRST I/O STATUS LONGWORD: STATUS CODE & BYTES XFERED
;       R1      - SECOND I/O STATUS LONGWORD: 0 FOR DISKS
;
;       THE I/O FUNCTION IS EXECUTED.
;
;       ALL REGISTERS EXCEPT R0-R4 ARE PRESERVED.
;
;--
DL_STARTIO:                                 ;START I/O OPERATION

;
;       COMPUTE PHYSICAL MEDIA ADDRESS
;
;               LBN = LBN * (SECTORS/BLOCK)
;               LBN/(SECTORS/TRACK) = D + SECTOR
;               D/(TRACKS/CYLINDER) = CYLINDER + TRACK
;

;
;       PREPROCESS UCB FIELDS
;
```

```
PREPROCESS:                                 ;
        MOVL    IRP$L_MEDIA(R3),-           ; Copy given MEDIA address (logical)
                UCB$L_MEDIA(R5)             ;  to the UCB.
        BBS     #IRP$V_PHYSIO,-             ;IF SET - PHYSICAL I/O
                IRP$W_STS(R3),10$
        MULL3   #2,UCB$L_MEDIA(R5),R0       ;SCALE LBN IN R0
        MOVZBL  UCB$B_SECTORS(R5),R2        ;GET NUMBER OF SECTORS PER TRACK
        CLRL    R1                          ;CLEAR HIGH PART OF DIVIDEND
        EDIV    R2,R0,R0,UCB$L_MEDIA(R5);CALCULATE SECTOR NUMBER AND STORE
        MOVZBL  UCB$B_TRACKS(R5),R2         ;GET NUMBER OF TRACKS PER CYLINDER
        EDIV    R2,R0,R0,R1                 ;CALCULATE TRACK AND CYLINDER
        MOVB    R1,UCB$L_MEDIA+1(R5)        ;STORE TRACK NUMBER
        MOVW    R0,UCB$L_MEDIA+2(R5)        ;STORE CYLINDER NUMBER
10$:
        MOVB    UCB$B_ERTMAX(R5),-          ;INITIALIZE ERROR RETRY COUNT
                UCB$B_ERTCNT(R5)            ;...
        MNEGW   UCB$W_BCNT(R5),UCB$W_BCR(R5)  ;INIT NEG BYTES LEFT TO XFER
        CLRW    UCB$W_DL_DPN(R5)            ;CLEAR DATA PATH NO. FOR USE AS-
                                            ;UBA RESOURCE ALLOCATION FLAG
        CLRB    UCB$B_DL_DPPE(R5)           ;CLEAR DATAPATH PURGE ERROR REGISTER
        MOVW    IRP$W_FUNC(R3),UCB$W_FUNC(R5)  ;SAVE FUNCTION CODE
        EXTZV   #IRP$V_FCODE,-              ;EXTRACT I/O FUNCTION CODE
                #IRP$S_FCODE,IRP$W_FUNC(R3),R1  ;...
        MOVB    R1,UCB$B_FEX(R5)            ;STORE FUNCTION DISPATCH INDEX
        CMPB    #IO$_SEEK,R1                ;SEEK FUNCTION?
        BNEQ    20$                         ;IF NEQ - NO
        MOVW    IRP$L_MEDIA(R3),-           ;STORE CYLINDER ADDRESS
                UCB$W_DC(R5)                ;...
20$:
        BICW    #UCB$M_DIAGBUF,-
                UCB$W_DEVSTS(R5)            ;CLR DIAGNOSTIC BUFFER PRESENT
        BBC     #IRP$V_DIAGBUF,-            ;IF CLR - NO DIAG BUFFER
                IRP$W_STS(R3),FDISPATCH ;...
        BISW    #UCB$M_DIAGBUF,UCB$W_DEVSTS(R5)  ;SET DIAG BUFFER PRESENT

;
;       CENTRAL FUNCTION DISPATCH
;

FDISPATCH:                                  ;FUNCTION DISPATCH
        MOVL    UCB$L_IRP(R5),R3            ;GET IRP ADDRESS
        BBS     #IRP$V_PHYSIO,-             ;IF SET - PHYSICAL I/O FUNCTION
                IRP$W_STS(R3),10$           ;...
        BBS     #UCB$V_VALID,-              ;IF SET - VOLUME SOFTWARE VALID
                UCB$W_STS(R5),10$           ;...
        MOVZWL  #SS$_VOLINV,R0              ;SET VOLUME INVALID STATUS
        BRW     RESETXFR                    ;RESET BYTE COUNT AND EXIT
10$:    CLRB    UCB$B_DL_DCHEK(R5)          ;CLEAR DATA CHECK IN PROGRESS
        MOVZBL  UCB$B_FEX(R5),R3            ;GET FUNCTION DISPATCH INDEX
        CASE    R3,<-                       ;DISPATCH TO FUNCTION HANDLING ROUTINE
                UNLOAD,-                    ; UNLOAD
                SEEK,-                      ; SEEK
                NOP,-                       ; RECALIBRATE (unsupported)
                DRVCLR,-                    ; DRVCLR
                NOP,-                       ; RELEASE PORT (unsupported)
                NOP,-                       ; OFFSET HEADS (unsupported)
                NOP,-                       ; RETURN TO CENTER (unsupported)
                PACKACK,-                   ; PACK ACKNOWLEDGE
                NOP,-                       ; SEARCH (unsupported)
                WRITECHECK,-                ; WRITE CHECK
                WRITEDATA,-                 ; WRITE DATA
                READDATA,-                  ; READ DATA
                NOP,-                       ; WRITE HEADER (unsupported)
                READHEAD,-                  ; READ HEADER
```

```
                NOP,-                   ; place holder
                NOP,-                   ; place holder
                AVAILABLE-              ; AVAILABLE
                >,LIMIT=#CDF_UNLOAD     ;

NOP:                                    ;NO-OP
SEEK:                                   ;SEEK
DRVCLR:                                 ;DRIVE CLEAR (GET STATUS & RESET)
DO_FUNCTION:
        EXFUNCL RETRYERR                ;EXECUTE FUNCTION - RETRY IF FAILURE
        BRB     NORMAL                  ;SUCCESSFUL - EXIT WITH NORMAL STATUS

PACKACK:                                ;PACK ACKNOWLEDGE (GET STATUS & RESET)
        BISW    #UCB$M_VALID, -         ;Set software volume valid bit.
                UCB$W_STS(R5)
        BRB     DO_FUNCTION             ;Then go do hardware function.

UNLOAD:                                 ;UNLOAD
AVAILABLE:                              ;AVAILABLE
        BICW    #UCB$M_VALID, -         ;Clear software volume valid bit.
                UCB$W_STS(R5)           ;and go complete operation without
        BRB     NORMAL                  ;any hardware interaction.

WRITECHECK:                             ;WRITE CHECK
READHEAD:                               ;READ HEADER
        BICW    #IO$M_DATACHECK,-       ;CLEAR DATA CHECK REQUEST-
                UCB$W_FUNC(R5)          ;TO PREVENT EXTRA WRITE CHECK

WRITEDATA:                              ;WRITE DATA
READDATA:                               ;READ DATA
        EXFUNCL RETRYERR,F_SEEK         ;EXECUTE EXPLICIT SEEK - RETRY IF FAIL

        MOVZBL  UCB$B_FEX(R5),R3        ;GET FUNCTION DISPATCH INDEX
        EXFUNCL RETRYERR                ;EXECUTE TRANSFER FUNCTION

;
;       OPERATON COMPLETION
;


NORMAL:                                 ;SUCCESSFUL OPERATION COMPLETE
        MOVZWL  #SS$_NORMAL,R0          ;SET NORMAL COMPLETION STATUS
        BRW     FUNCXT                  ;FUNCTION EXIT

RETRYERR:                               ;RETRIABLE ERROR
        DECB    UCB$B_ERTCNT(R5)        ;ANY RETRIES LEFT?
        BEQL    FATALERR                ;IF EQL - NO
        BRW     FDISPATCH               ;RETRY FUNCTION

FATALERR:                               ;UNRECOVERABLE ERROR
        MOVZWL  #SS$_VOLINV,R0          ;ASSUME VOLUME INVALID STATUS
        BBS     #RL_MP_V_VC,-           ;IF SET - VOLUME INVALID
                UCB$W_DL_MP(R5),FUNCXT  ;...

        MOVZWL  #SS$_WRITLCK,R0         ;ASSUME WRITE LOCK ERROR STATUS
        BBC     #RL_MP_V_WL,-           ;IF CLR - VOLUME NOT WRITE LOCKED
                UCB$W_DL_MP(R5),5$      ;...
        BBS     #RL_MP_V_WGE,-          ;IF SET - WRITE GATE ERROR
                UCB$W_DL_MP(R5),FUNCXT  ;IF WL & WGE SET - WRITE LOCK ERROR

5$:     MOVZWL  #SS$_DATACHECK,R0       ;ASSUME DATA CHECK ERROR STATUS
        TSTB    UCB$B_DL_DCHEK(R5)      ;WRITE CHECK IN PROGRESS?
        BEQL    10$                     ;IF EQL - NO
        BBS     #RL_CS_V_OPI,-          ;IF SET - NOT WRITE CHECK ERROR
                UCB$W_DL_CS(R5),10$     ;...
        BBS     #RL_CS_V_CRC,-          ;IF SET - WRITE CHECK ERROR
                UCB$W_DL_CS(R5),FUNCXT  ;...
```

```
10$:    MOVZWL  #SS$_PARITY,R0          ;ASSUME PARITY ERROR STATUS
        BBS     #RL_CS_V_CRC,-          ;IF SET - CRC ERROR
                UCB$W_DL_CS(R5),FUNCXT  ;OR DATAPATH PURGE ERROR

20$:    MOVZWL  #SS$_DRVERR,R0          ;ASSUME DRIVE ERROR STATUS
        BBS     #RL_CS_V_DE,-           ;IF SET - DRIVE ERROR
                UCB$W_DL_CS(R5),FUNCXT  ;...

        MOVZWL  #SS$_CTRLERR,R0         ;ASSUME CONTROLLER ERROR STATUS

FUNCXT:                                 ;FUNCTION EXIT
        PUSHL   R0                      ;SAVE FINAL REQUEST STATUS
        JSB     G^IOC$DIAGBUFILL        ;FILL DIAGNOSTIC BUFFER IF PRESENT
        CMPB    #CDF_WRITECHECK,UCB$B_FEX(R5)  ;DRIVE RELATED FUNCTION?
        BGTRU   10$                     ;IF GTRU - YES
        CMPB    #CDF_AVAILABLE,UCB$B_FEX(R5)   ;DRIVE RELATED FUNCTION?
        BEQL    10$                     ;IF EQL  - YES
        MOVL    UCB$L_IRP(R5),R3        ;RETRIEVE ADDRESS OF IRP
        ADDW3   UCB$W_BCR(R5),-         ;CALCULATE BYTES TRANSFERRED
                IRP$W_BCNT(R3),2(SP)    ;...
        TSTW    UCB$W_DL_DPN(R5)        ;ARE UBA RESOURCES ALLOCATED?
        BEQL    20$                     ;IF EQL - NO
        BBC     #UCB$V_DL_MAPPING,-     ;ADAPTER MAPPING?
                UCB$W_DL_FLAGS(R5),10$  ;IF BC NO
        RELDPR                          ;RELEASE DATA PATH
        RELMPR                          ;RELEASE MAP REGISTERS
        BRB     20$                     ;JOIN COMMON CODE
10$:    MOVL    UCB$L_DL_SVAPTE(R5),-   ;RESTORE ORIGINAL SVAPTE
                UCB$L_SVAPTE(R5)        ;
20$:    RELCHAN                         ;RELEASE CHANNEL IF OWNED

        CLRL    R1                      ;CLEAR SECOND STATUS LONGWORD
        POPL    R0                      ;RETRIEVE FINAL REQUEST STATUS
        REQCOM                          ;COMPLETE REQUEST
        .PAGE
;
; FEXL - RL11 HARDWARE FUNCTION EXECUTION
;
; THIS ROUTINE IS CALLED VIA A BSB WITH A BYTE IMMEDIATELY FOLLOWING THAT
; SPECIFIES THE ADDRESS OF AN ERROR ROUTINE. ALL DATA IS ASSUMED TO HAVE BEEN
; SET UP IN THE UCB BEFORE THE CALL. THE APPROPRIATE PARAMETERS ARE LOADED
; INTO DEVICE REGISTERS AND THE FUNCTION IS INITIATED. THE RETURN ADDRESS
; IS STORED IN THE UCB AND A WAIT FOR INTERRUPT IS EXECUTED. WHEN THE
; INTERRUPT OCCURS, CONTROL IS RETURNED TO THE CALLER.
;
; INPUTS:
;
;       R3 = FUNCTION TABLE DISPATCH INDEX
;       R5 = DEVICE UNIT UCB ADDRESS
;
;       00(SP) = RETURN ADDRESS OF CALLER
;       04(SP) = RETURN ADDRESS OF CALLER'S CALLER
;
;       IMMEDIATELY FOLLOWING INLINE AT THE CALL SITE IS A BYTE WHICH CONTAINS
;       A BRANCH DESTINATION TO AN ERROR RETRY ROUTINE.
;
; OUTPUTS:
;
;       THERE ARE FOUR EXITS FROM THIS ROUTINE:
;
;       1. SPECIAL CONDITION - THIS EXIT IS TAKEN IF A POWER FAILURE OCCURS
;               OR THE OPERATION TIMES OUT. IT IS A JUMP TO THE APPROPRIATE
;               ERROR ROUTINE.
;
;       2. FATAL ERROR - THIS EXIT IS TAKEN IF A FATAL CONTROLLER OR DRIVE
```

```
;               ERROR OCCURS OR IF ANY ERROR OCCURS AND ERROR RETRY IS EITHER
;               INHIBITED OR EXHAUSTED. IT IS A JUMP TO THE FATAL ERROR EXIT
;               ROUTINE.
;
;       3. RETRIABLE ERROR - THIS EXIT IS TAKEN IF A RETRIABLE CONTROLLER
;               OR DRIVE ERROR OCCURS AND ERROR RETRY IS NEITHER INHIBITED
;               NOR EXHAUSTED. IT CONSISTS OF TAKING THE ERROR BRANCH EXIT
;               SPECIFIED AT THE CALL SITE.
;
;       4. SUCCESSFUL OPERATION - THIS EXIT IS TAKEN IF NO ERRORS OCCUR
;               DURING THE OPERATION. IT CONSISTS OF A RETURN INLINE.
;
;       IN ALL CASES IF AN ERROR OCCURS, AN ATTEMPT IS MADE TO LOG THE ERROR.
;
;       IN ALL CASES FINAL DEVICE REGISTERS ARE RETURNED VIA THE UCB.
;
;       UCB$W_BCR(R5) = NEGATIVE BYTES REMAINING TO TRANSFER

        .PAGE
FEXL:                                           ;FUNCTION EXECUTOR
        POPL    UCB$L_DPC(R5)                   ;SAVE DRIVER PC VALUE
        MOVB    R3,UCB$B_CEX(R5)                ;SAVE CASE INDEX
        MOVL    UCB$L_CRB(R5),R0                ;GET ADDRESS OF PRIMARY CRB
        MOVL    CRB$L_INTD+VEC$L_IDB(R0),R1  ;GET ADDRESS OF IDB
        CMPL    R5,IDB$L_OWNER(R1)              ;DOES THIS PROCESS OWN CHANNEL?
        BNEQ    10$                             ;IF NEQ - NO
        MOVL    IDB$L_CSR(R1),R4                ;SET ASSIGNED CHANNEL CSR ADDRESS
        BRB     20$                             ;
10$:    REQPCHAN                                ;REQUEST CHANNEL (RETURNS R4 = CSR ADR)
20$:    CASE    R3,<-                           ;DISPATCH TO PROPER FUNCTION ROUTINE
                IMMED,-                         ;NO OPERATION
                IMMED,-                         ;UNLOAD VOLUME (NOP)
                POSIT,-                         ;SEEK CYLINDER
                IMMED,-                         ;RECALIBRATE (NOP)
                DRCLR,-                         ;DRIVE CLEAR (GET STATUS & RESET)
                IMMED,-                         ;RELEASE DRIVE (NOP)
                IMMED,-                         ;OFFSET HEADS (NOP)
                IMMED,-                         ;RETURN TO CENTERLINE (NOP)
                DRCLR,-                         ;PACK ACKNOWLEDGE
                IMMED,-                         ;SEARCH (NOP)
                >                               ;
        BRW     XFER                            ;TRANSFER FUNCTION
        .PAGE
;
; IMMEDIATE FUNCTION EXECUTION
;
;       FUNCTIONS INCLUDE:
;
;               NO OPERATION,
;               DRIVE CLEAR, AND
;               PACK ACKNOWLEDGE
;
; INPUTS:
;       R3      - CASE INDEX
;       R4      - CSR ADDRESS
;       R5      - UCB ADDRESS
;
; FUNCTIONAL DESRIPTION:
;
; INTERRUPTS ARE LOCKED OUT, THE APPROPRIATE FUNCTION IS INITIATED WITH
; INTERRUPT ENABLE, AND A WAIT FOR INTERRUPT AND KEEP CHANNEL IS EXECUTED.
;
```

```
DRCLR:                                  ;DRIVE CLEAR
        BISW    #RL_DA_M_STS!-          ;SET GETSTATUS,RESET,AND MARK IN DAR
                RL_DA_M_RST!RL_DA_M_MRK,RL_DA(R4)  ;...

IMMED:                                  ;IMMEDIATE FUNCTION EXECUTION
        CKPWR   SAVE_R0=NO              ;DISABLE INTERRUPTS, CHECK POWER,-
                                        ;AND PUT UNIT NUMBER IN R2<9:8>
        BISW3   R2,FTAB[R3],RL_CS(R4)   ;MERGE UNIT WITH FNTN AND EXECUTE
        WFIKPCH RETREG,#2               ;WAIT FOR INTERRUPT
        IOFORK                          ;RETURN FROM ISR-
                                        ;CREATE FORK PROCESS (&JSB BACK TO ISR)
        BRW     RETREG                  ;
        .PAGE
;
; POSITIONING FUNCTION EXECUTION
;
;       FUNCTIONS INCLUDE:
;
;               SEEK CYLINDER
;
; INPUTS:
;       R3      - CASE INDEX
;       R4      - DEVICE CSR ADDRESS
;       R5      - UCB ADDRESS
;
; FUNCTIONAL DESRIPTION:
;
; THE CYLINDER DIFFERENCE WORD IS CALCULATED AND LOADED INTO THE DISK
; ADDRESS REGISTER, INTERRUPTS ARE LOCKED OUT, AND THE SEEK FUNCTION
; IS INITIATED WITHOUT INTERRUPT ENABLE. THE CONTROLLER IS THEN POLLED
; FOR READY, AND DEVICE INTERRUPTS ARE ENABLED.
;
; SINCE THE RL01/RL02 DO NOT ISSUE AN INTERRUPT UPON COMPLETION OF A
; SEEK, OVERLAPPED SEEKS ARE NOT ATTEMPTED, AND ONE OF THE FOLLOWING IS
; PERFORMED.
;
;       IF ONLY A SEEK FUNCTION IS BEING REQUESTED, A DUMMY READ HEADER
;       FUNCTION IS ISSUED AND A WAITFOR INTERRUPT IS INITIATED.
;       THE READ HEADER IS USED TO SIGNAL THE END OF THE SEEK, SINCE IT
;       WILL ISSUE AN INTERRUPT SHORTLY (315 USEC AVG) AFTER THE SEEK IS
;       COMPLETE. IT WILL ALSO SENSE FOR A TIMEOUT DURING THE SEEK.
;
;       IF THE SEEK IS ASSOCIATED WITH A DATA TRANSFER REQUEST (RL01/RL02
;       TRANSFER FUNCTIONS REQUIRE EXPLICIT SEEKS), THE PROGRAM KEEPS THE
;       CHANNEL AND RETURNS TO FDISPATCH TO ISSUE THE TRANSFER REQUEST
;       WHILE THE SEEK IS STILL IN PROGRESS. WHEN THE SEEK COMPLETES, THE
;       RL11 CONTROLLER WILL BEGIN THE TRANSFER.
;
```

```
POSIT:                                          ;POSITIONING FUNCTION
;
; OBTAIN CURRENT DISK ADDRESS
;
; IF THERE HAS NOT BEEN A PREVIOUS TRANSFER DURING THIS REQUEST,
; A READ HEADER IS EXECUTED TO DETERMINE THE CURRENT DISK ADDRESS.
;
        TSTW    UCB$W_DL_DPN(R5)        ;WAS THERE A PREVIOUS TRANSFER?
        BEQL    10$                     ;IF EQL - NO, READ HEADER
        BICW3   #^O77,UCB$W_DL_DA(R5),R1  ;PUT CURRENT CYL & SURFACE IN R1
        BRW     60$                     ;CALCULATE DIFFERENCE WORD
5$:     BRW     50$                     ;CONTINUE
10$:    MOVZBL  #8,R3                   ;SET READ HEADER RETRY COUNT IN R3
20$:    CKPWR   SAVE_R0=NO              ;DISABLE INTERRUPTS, CHECK POWER,-
                                        ;AND PUT UNIT NUMBER IN R2<9:8>
        BISW3   R2,#F_READHEAD!RL_CS_M_IE,-  ;EXECUTE READ HEADER
                RL_CS(R4)               ;...
        WFIKPCH 40$,#2                  ;WAIT FOR INTERRUPT OR TIMEOUT
        IOFORK                          ;CREATE FORK PROCESS
        BBC     #RL_CS_V_CE,UCB$W_DL_CS(R5),5$ ;BR ON NO ERRORS
        DECB    R3                      ;DECREMENT READ HEADER RETRY COUNT
        BNEQ    20$                     ;IF NEQ - RETRY READ HEADER
                                        ;IF EQL - READ HEADER RETRY EXHAUSTED -
                                        ;TRY PREVIOUS TRACK
        MOVZBW  #^X80!RL_DA_M_MRK,-     ;LOAD REVERSE SEEK DIFFERENCE WORD
                RL_DA(R4)               ;...
        CKPWR   SAVE_R0=NO              ;DISABLE INTERRUPTS, CHECK POWER,-
                                        ;AND PUT UNIT NUMBER IN R2<9:8>
        BISW3   R2,#F_SEEK!RL_CS_M_IE,- ;EXECUTE REVERSE SEEK
                RL_CS(R4)               ;...
        WFIKPCH 40$,#2                  ;WAIT FOR SEEK TO BEGIN (INTERRUPT)
        IOFORK                          ;CREATE FORK PROCESS
        CKPWR   SAVE_R0=NO              ;DISABLE INTERRUPTS, CHECK POWER,-
                                        ;AND PUT UNIT NUMBER IN R2<9:8>
        BISW3   R2,#F_READHEAD!RL_CS_M_IE,-  ;TRY READ HEADER ON NEW TRACK
                RL_CS(R4)               ;...
        WFIKPCH 40$,#2                  ;WAITFOR INTERRUPT OR TIMEOUT
        IOFORK                          ;CREATE FORK PROCESS
        BBC     #RL_CS_V_CE,UCB$W_DL_CS(R5),50$ ;BR IF NO HEADER ERROR
40$:                                    ;CANNOT READ CURRENT DISK ADDRESS
;       CLRB    UCB$B_ERTCNT(R5)        ;CLEAR RETRY COUNT
        BRW     RETREG                  ;
50$:                                    ;FOUND CURRENT DISK ADDRESS
        BICW3   #^O77,UCB$W_DL_MP(R5),R1  ;PUT CURRENT CYL & SURFACE IN R1

;
; CALCULATE CYLINDER DIFFERENCE WORD
;
60$:    CLRL    R0                      ;CLEAR R0 FOR DESIRED ADDRESS
        INSV    UCB$W_DA+1(R5),#6,#1,R0 ;INSERT DESIRED SURFACE IN R0<6>
        INSV    UCB$W_DC(R5),#7,#9,R0   ;INSERT DESIRED CYLINDER IN R0<15:7>
        CMPW    R0,R1                   ;IS A SEEK NEEDED?
        BEQL    80$                     ;IF EQL - NO
        BICB    #^O177,R1               ;REMOVE SURFACE BIT
        BICB    #^O177,R0               ;REMOVE SURFACE BIT
        SUBW    R0,R1                   ;SUBTRACT DESIRED FROM ACTUAL
        BEQL    70$                     ;IF EQL - ONLY CHANGE SURFACE
        BCC     70$                     ;IF CC - ACTUAL>=DESIRED
        MNEGW   R1,R1                   ;ACTUAL<DESIRED, MAKE POSITIVE DIFF
        BISW    #4,R1                   ;SET SIGN FOR MOVE TO CENTER OF DISK
70$:    INSV    UCB$W_DA+1(R5),#4,#1,R1 ;INSERT SURFACE BIT
        BISW3   #RL_DA_M_MRK,R1,RL_DA(R4)  ;SET MARKER AND LOAD DIFFERENCE WORD
```

```
;
; EXECUTE SEEK
;
        CKPWR   SAVE_R0=NO              ;DISABLE INTERRUPTS, CHECK POWER,-
                                        ;AND PUT UNIT NUMBER IN R2<9:8>
        BISW3   R2,#F_SEEK!RL_CS_M_IE,- ;EXECUTE SEEK FUNCTION
                RL_CS(R4)               ;...
        WFIKPCH 40$,#2                  ;WAIT FOR SEEK TO BEGIN (INTERRUPT)
        IOFORK                          ;CREATE FORK PROCESS
80$:    CMPB    #IO$_SEEK,UCB$B_FEX(R5) ;IS SEEK ASSOCIATED WITH A TRANSFER?
        BEQL    90$                     ;IF EQL - NO, SEEK ONLY

;
; RETURN FOR SEEK ASSOCIATED WITH A TRANSFER REQUEST
;
        INCL    UCB$L_DPC(R5)           ;ADJUST TO CORRECT RETURN ADDRESS
        JMP     @UCB$L_DPC(R5)          ;RETURN TO DRIVER FOR TRANSFER
;
; RETURN FOR SEEK ONLY REQUEST
;
90$:    CKPWR   SAVE_R0=NO              ;DISABLE INTERRUPTS, CHECK POWER,-
                                        ;AND PUT UNIT NUMBER IN R2<9:8>
        BISW3   R2,#F_READHEAD!RL_CS_M_IE,-  ;EXECUTE DUMMY READ HEADER
                RL_CS(R4)               ;...
        WFIKPCH RETREG,#2               ;WAIT FOR SEEK TO COMPLETE (INTERRUPT)
        IOFORK                          ;CREATE FORK PROCESS
        BRW     RETREG                  ;
        .PAGE

;
; TRANSFER FUNCTION EXECUTION
;
;       FUNCTIONS INCLUDE:
;
;               WRITE CHECK
;               WRITE DATA
;               READ DATA, AND
;               READ HEADER
;
; INPUTS:
;       R3      - CASE INDEX
;       R4      - DEVICE CSR ADDRESS
;       R5      - UCB ADDRESS
;
; FUNCTIONAL DESCRIPTION:
;
; A UNIBUS DATAPATH IS REQUESTED FOLLOWED BY THE APPROPRIATE NUMBER OF MAP
; REGISTERS REQUIRED FOR THE TRANSFER. THE TRANSFER PARAMETERS ARE LOADED
; INTO THE DEVICE REGISTERS, INTERRUPTS ARE LOCKED OUT, THE FUNCTION IS
; INITIATED, AND A WAITFOR INTERRUPT AND KEEP CHANNEL IS EXECUTED.
;
; UPON RETURN FROM THE INTERRUPT SERVICE ROUTINE, IF THE TRANSFER IS
; COMPLETE, THE APPROPRIATE EXIT IS TAKEN. IF THE FUNCTION IS NOT COMPLETE
; TRANSFER PARAMETERS ARE UPDATED AND A RETURN TO FDISPATCH IS EXECUTED TO
; REISSUE SEEK AND TRANSFER FUNCTIONS WHILE KEEPING CHANNEL AND UBA
; RESOURCES. IF A DATA CHECK HAS BEEN REQUESTED, IT IS PERFORMED
; BEFORE RETURNING TO FDISPATCH.
;
```

```
XFER:                                       ;TRANSFER FUNCTION EXECUTION
        BBS     #UCB$V_DL_MAPPING,-         ;ADAPTER MAPPING?
                UCB$W_DL_FLAGS(R5),2$       ;BRANCH IF ADAPTER MAPPING.
        MOVW    UCB$A_DL_BUF_PA(R5),UCB$W_DL_SBA(R5);GET 1ST WORD OF BUFFER ADDR
        MOVZWL  UCB$A_DL_BUF_PA+2(R5),R0;GET BITS 16:21 OF BUFFER ADDRESS
        MOVW    R0,RL_BAE(R4)               ;SET MEMORY EXTENSION BITS IN BAE
        ASHL    #4,R0,R0                    ;PUT MEMORY EXTENSION BITS IN <5:4>
        MOVB    R0,UCB$B_DL_XBA(R5)         ;OF CSR
;
; FIRST TRANSFER OF THIS I/O REQUEST - ALLOCATE RESOURCES
;
        TSTW    UCB$W_DL_DPN(R5)            ;RESOURCES ALREADY ALLOCATED?
        BNEQ    5$                          ;IF NEQ - YES
        CLRL    UCB$A_DL_MOVRTN(R5)         ;ASSUME READ
        CMPB    #CDF_WRITEDATA,R3           ;WRITE DATA?
        BNEQ    1$                          ;IF NEQ NO
        MOVAB   G^IOC$MOVFRUSER,-           ;SET MOVE ROUTINE ADDRESS FOR
                UCB$A_DL_MOVRTN(R5)         ;1ST PARTIAL WRITE
1$:     MOVL    UCB$L_SVAPTE(R5),UCB$L_DL_SVAPTE(R5);SAVE SVAPTE FOR BUFFER COPY
        MNEGW   #1,UCB$W_DL_DPN(R5)         ;SET FIRST XFER FLAG
        BRB     5$                          ;JOIN COMMON CODE

;
; FIRST TRANSFER OF THIS I/O REQUEST - ALLOCATE RESOURCES
;
2$:     TSTW    UCB$W_DL_DPN(R5)            ;UBA RESOURCES ALREADY ALLOCATED?
        BNEQ    5$                          ;IF NEQ - YES
        REQDPR                              ;REQUEST DATAPATH
        REQMPR                              ;REQUEST MAP REGISTERS
        LOADUBA                             ;LOAD UNIBUS MAP REGISTERS
        MOVL    UCB$L_CRB(R5),R1            ;GET CRB ADDRESS
        EXTZV   #VEC$V_DATAPATH,#VEC$S_DATAPATH,-  ;EXTRACT DATAPATH NUMBER -
                CRB$L_INTD+VEC$B_DATAPATH(R1),R0   ;FOR UBA RESOURCE FLAG
        MOVW    R0,UCB$W_DL_DPN(R5)         ;INDICATE UBA RESOURCES ALLOCATED

        MOVZWL  UCB$W_BOFF(R5),R0           ;GET BYTE OFFSET IN PAGE
        INSV    CRB$L_INTD+VEC$W_MAPREG(R1),-  ;INSERT HIGH 7 BITS OF ADDRESS
                #9,#7,R0                    ;...
        MOVW    R0,UCB$W_DL_SBA(R5)         ;SET BUFFER ADDRESS
        EXTZV   #7,#2,CRB$L_INTD+VEC$W_MAPREG(R1),R0  ;GET MEMORY EXTENSION BITS
        MULB3   #16,R0,UCB$B_DL_XBA(R5) ;POSITION MEMORY EXTENSION BITS TO <5:4>

;
; COMMON TRANSFER POINT
;

;
; FOR A READ OPERATION WHEN NO ADAPTER MAPPING IS PRESENT EMPTY THE
; INTERNAL PHYSICALLY CONTIGUOUS BUFFER FROM THE PREVIOUS READ TO THE
; USER'S BUFFER.
;
5$:     BSBW    DL_MOVE_TO_BUFFER           ;COPY TO USER BUFFER
;
; PUT BUFFER ADDRESS, WORD COUNT, AND DISK ADDRESS IN DEVICE REGISTERS
;
```

```
        MOVW    UCB$W_DL_SBA(R5),RL_BA(R4)  ;SET BUFFER ADDRESS
        MNEGW   UCB$W_BCR(R5),-             ;GET BYTES LEFT TO TRANSFER AND -
                UCB$W_DL_PBCR(R5)           ;ASSUME ONLY ONE TRANSFER NEEDED
        MOVZBL  UCB$B_SECTORS(R5),R2        ;GET SECTORS/SURFACE
        MOVZBL  UCB$W_DA(R5),R1             ;GET DESIRED SECTOR
        SUBW    R1,R2                       ;CALCULATE SECTORS LEFT ON SURFACE
        MULW    #256,R2                     ;CONVERT TO BYTES LEFT ON SURFACE
        CMPW    UCB$W_DL_PBCR(R5),R2        ;ARE ADDITIONAL TRANSFERS REQUIRED?
        BLEQU   10$                         ;IF LEQU - NO
        MOVW    R2,UCB$W_DL_PBCR(R5)        ;SET BYTE COUNT FOR THIS TRANSFER
;
; FOR A WRITE OPERATION WHEN NO ADAPTER MAPPING IS PRESENT
; FILL INTERNAL PHYSICALLY CONTIGUOUS BUFFER FROM THE USER'S BUFFER.
;
10$:    BSBW    DL_MOVE_FROM_BUFFER         ;COPY FROM USER BUFFER

        MOVZBL  UCB$B_DL_XBA(R5),R0         ;SET MEMORY EXTENSION BITS
        BISW    FTAB[R3],R0                 ;MERGE XBA BITS WITH FUNCTION
        DIVW3   #2,UCB$W_DL_PBCR(R5),R2     ;CALCULATE TRANSFER WORD COUNT
        MNEGW   R2,RL_MP(R4)                ;SET TRANSFER WORD COUNT

        MOVZBL  UCB$W_DA(R5),R1             ;PUT DESIRED SECTOR IN R1<5:0>
        INSV    UCB$W_DA+1(R5),#6,#1,R1     ;INSERT DESIRED SURFACE IN R1<6>
        INSV    UCB$W_DC(R5),#7,#9,R1       ;INSERT DESIRED CYLINDER IN R1<15:7>
        MOVW    R1,RL_DA(R4)                ;SET DESIRED DISK ADDRESS

;
; EXECUTE THE TRANSFER FUNCTION
;

        CKPWR                               ;DISABLE INTERRUPTS, CHECK POWER,-
                                            ;AND PUT UNIT NUMBER IN R2<9:8>
        BISW3   R2,R0,RL_CS(R4)             ;EXECUTE FUNCTION
        WFIKPCH RETREG,#6                   ;WAITFOR INTERRUPT AND KEEP CHANNEL
                                            ;RETURN HERE FROM ISR SAVING REGISTERS
        IOFORK                              ;CREATE FORK PROCESS (RETURN TO ISR)
                                            ;RETURN HERE FROM ISR REI ROUTINE

;
; PURGE DATAPATH
;

        CLRB    UCB$B_DL_DPPE(R5)           ;CLEAR DATAPATH PURGE ERROR
        JSB     G^IOC$PURGDATAP             ;PURGE DATAPATH
        BLBS    R0,20$                      ;IF SET - NO PURGE ERRORS
        INCB    UCB$B_DL_DPPE(R5)           ;SET DATAPATH PURGE ERROR

;
; SAVE UBA REGISTERS FOR UPDATE AND REGDUMP ROUTINES
;
```

```
20$:    BBC     #UCB$V_DL_MAPPING,-     ;ADAPTER MAPPING?
                UCB$W_DL_FLAGS(R5),30$  ;IF BC NO
        MOVL    R1,UCB$L_DL_DPR(R5)     ;SAVE DATAPATH REGISTER
        EXTZV   #9,#7,UCB$W_DL_BA(R5),R0  ;EXTRACT LOW BITS OF FINAL MAP REG NO.
        EXTZV   #4,#2,UCB$W_DL_CS(R5),R1  ;EXTRACT HI BITS OF FINAL MAP REG NO.
        INSV    R1,#7,#2,R0             ;INSERT HIGH BITS OF FINAL MAP REGISTER
        CMPW    #495,R0                 ;LEGAL MAP REGISTER NUMBER?
        BGEQ    25$                     ;IF GEQ - YES
        MOVZWL  #495,R0                 ;RESTRICT MAP REGISTER NUMBER
25$:    MOVL    (R2)[R0],UCB$L_DL_FMPR(R5)  ;SAVE FINAL MAP REGISTER NUMBER
        CLRL    UCB$L_DL_PMPR(R5)       ;CLEAR PREVIOUS MAP REGISTER CONTENTS
        DECL    R0                      ;CALCULATE PREVIOUS MAP REGISTER NUMBER
        CMPV    #VEC$V_MAPREG,#VEC$S_MAPREG,-  ;ANY PREVIOUS MAP REGISTER?
                CRB$L_INTD+VEC$W_MAPREG(R3),R0 ;...
        BGTR    30$                     ;IF GTR - NO
        MOVL    (R2)[R0],UCB$L_DL_PMPR(R5)  ;SAVE PREVIOUS MAP REGISTER
30$:    BBC     #RL_CS_V_CE,UCB$W_DL_CS(R5),40$  ;IF CLR - NO RL ERRORS
        BRW     RETREG                  ;DEVICE ERROR
40$:    BLBC    UCB$B_DL_DPPE(R5),45$   ;IF CLR - NO PURGE ERROR
        BRW     RETREG                  ;PURGE ERROR

;
; RETURN HEADER INFORMATION FOR READ HEADER FUNCTION
;
45$:    CMPB    #CDF_READHEAD,UCB$B_CEX(R5)  ;READ HEADER FUNCTION?
        BNEQ    DATACHECK               ;IF NEQ - NO
        PUSHL   UCB$W_BCR(R5)           ;SAVE NEG BYTES REMAINING
        PUSHL   UCB$L_SVAPTE(R5)        ;SAVE ADDRESS OF PTE
        MOVAB   UCB$W_DL_DB(R5),R1      ;SET ADDRESS OF INTERNAL BUFFER
        MOVL    #6,R2                   ;SET NUMBER OF BYTES TO MOVE
        CMPW    R2,UCB$W_BCNT(R5)       ;ROOM FOR FULL HEADER?
        BLSSU   50$                     ;IF LSSU - YES
        MOVZWL  UCB$W_BCNT(R5),R2       ;SET LENGTH OF PARTIAL HEADER
50$:    SUBW3   UCB$W_BCNT(R5),R2,UCB$W_BCR(R5)  ;CALCULATE TRANSFER BYTE COUNT
        JSB     G^IOC$MOVTOUSER         ;MOVE HEADER TO USER BUFFER
        POPL    UCB$L_SVAPTE(R5)        ;RESTORE ADDRESS OF PTE
        POPL    UCB$W_BCR(R5)           ;RESTORE NEG BYTES REMAINING

;
; PERFORM DATA CHECK, IF REQUESTED
;
DATACHECK:                              ;DATACHECK AFTER PARTIAL TRANSFER
        BBC     #IO$V_DATACHECK,-       ;IF CLR - DATA CHECK NOT REQUESTED
                UCB$W_FUNC(R5),UPDATE   ;...
        BBSC    #0,UCB$B_DL_DCHEK(R5),- ;IF SET - DATA CHECK ALREADY PERFORMED
                UPDATE                  ;...
        INCB    UCB$B_DL_DCHEK(R5)      ;SET DATA CHECK IN PROGRESS
        MOVZBL  #IO$_WRITECHECK,R3      ;SET CASE INDEX TO WRITE CHECK
        BRW     XFER                    ;BRANCH TO PERFORM WRITE CHECK

;
; UPDATE BUFFER ADDRESS, CURRENT DISK ADDRESS, AND BYTES REMAINING
; FOR NEXT TRANSFER
;
UPDATE:                                 ;UPDATE TRANSFER PARAMETERS
        BBC     #UCB$V_DL_MAPPING,-     ;ADAPTER MAPPING?
                UCB$W_DL_FLAGS(R5),10$  ;IF BC NO
        BICB3   #^XCF,UCB$W_DL_CS(R5),- ;SAVE MEMORY EXTENSION BITS
                UCB$B_DL_XBA(R5)        ;...
        MOVW    UCB$W_DL_BA(R5),-       ;UPDATE SAVED BUFFER ADDRESS
                UCB$W_DL_SBA(R5)        ;...
```

```
10$:    CLRB    UCB$W_DA(R5)            ;UPDATE DESIRED SECTOR TO ZERO
        ADDL3   #^O100,UCB$W_DL_DA(R5),R1  ;INCREMENT CYLINDER & SURFACE
        EXTZV   #6,#1,R1,R2             ;EXTRACT DESIRED DISK SURFACE
        MOVB    R2,UCB$W_DA+1(R5)       ;UPDATE DESIRED DISK SURFACE
        EXTZV   #7,#9,R1,R2             ;EXTRACT DESIRED DISK CYLINDER
        MOVW    R2,UCB$W_DC(R5)         ;UPDATE DESIRED DISK CYLINDER
        ADDW    UCB$W_DL_PBCR(R5),-     ;UPDATE NEG BYTES REMAINING TO XFER
                UCB$W_BCR(R5)           ;...
        BEQL    RETREG                  ;IF EQL - TRANSFER COMPLETE
        BRW     FDISPATCH               ;MORE BYTES REMAINING - CONTINUE

;
; GET STATUS AND RESET ERRORS
;
RETREG:                                 ;GET STATUS AND RESET ERRORS
;
; FOR A READ OPERATION WHEN NO ADAPTER MAPPING IS PRESENT
; EMPTY INTERNAL BUFFER INTO USER'S BUFFER FOR LAST READ
;
        BSBW    DL_MOVE_TO_BUFFER       ;MOVE LAST READ INTO USER'S BUFFER

        BITW    #UCB$M_TIMOUT!UCB$M_POWER,- ; TIMEOUT OR POWERFAIL?
                UCB$L_STS(R5)           ;
        BEQL    0$                      ;BR IF NO
        IOFORK                          ;ELSE, FORK
0$:     MOVW    #RL_DA_M_STS!-          ;PUT GET STATUS IN DAR
                RL_DA_M_MRK,RL_DA(R4)   ;...
        CLRL    R2                      ;CLEAR R2 FOR UNIT NUMBER
        INSV    UCB$W_UNIT(R5),#8,#8,R2 ;GET UNIT NUMBER
        BISW3   R2,#F_GETSTATUS,RL_CS(R4)  ;EXECUTE GET STATUS
        DEVICELOCK -
                LOCKADDR=UCB$L_DLCK(R5),- ;LOCK DEVICE ACCESS
                LOCKIPL=UCB$B_DIPL(R5),- ;RAISE IPL
                SAVIPL=-(SP),-          ;SAVE CURRENT IPL
                PRESERVE=NO             ;DON'T PRESERVE R0
        BSBW    DL_WAIT                 ;WAIT FOR CONTROLLER
        MOVW    RL_MP(R4),UCB$W_DL_MP(R5)  ;RETRIEVE ERROR REGISTER
        MOVW    #RL_DA_M_RST!-          ;PUT GET STATUS & RESET IN DAR
                RL_DA_M_STS!RL_DA_M_MRK,RL_DA(R4)  ;...
        BISW3   R2,#F_GETSTATUS,RL_CS(R4)  ;EXECUTE RESET
        BSBW    DL_WAIT                 ;WAIT FOR CONTROLLER
        DEVICEUNLOCK -
                LOCKADDR=UCB$L_DLCK(R5),- ;UNLOCK DEVICE ACCESS
                NEWIPL=(SP)+,-          ;RESTORE IPL
                PRESERVE=NO             ;DON'T PRESERVE R0

;
; DETERMINE EXIT - SPECIAL CONDITION, FATAL ERROR, RETRIABLE ERROR, OR SUCCESS
;
        CMPZV   #0,#5,UCB$W_DL_MP(R5),- ;HEADS, BRUSHES, STATE OK?
                #RL_MP_M_BH!RL_MP_M_HO!RL_SLM  ;...
        BEQL    1$                      ;IF EQL - YES, ONLINE
        BICW    #UCB$M_TIMOUT,UCB$W_STS(R5)  ;CLEAR DEVICE TIME OUT
        MOVZWL  #SS$_MEDOFL,R0          ;SET MEDIUM OFFLINE STATUS
        BRW     FUNCXT                  ;RETURN
1$:     BITW    #UCB$M_POWER!-          ;POWER FAIL OR DEVICE TIMEOUT?
                UCB$M_TIMOUT,UCB$W_STS(R5)  ;...
        BNEQ    SPECOND                 ;IF NEQ - YES, SPECIAL CONDITION
```

```
        BBS     #RL_MP_V_VC,UCB$W_DL_MP(R5),20$  ;IF SET - VOLUME INVALID
        BBS     #RL_CS_V_CE,UCB$W_DL_CS(R5),2$  ;IF SET - RL ERROR
        BLBC    UCB$B_DL_DPPE(R5),10$   ;IF CLR - NO PURGE ERROR
2$:     JSB     G^ERL$DEVICERR          ;ALLOCATE AND FILL ERROR MESSAGE BUFFER
        BBS     #IO$V_INHRETRY,UCB$W_FUNC(R5),20$  ;IF SET - RETRY INHIBITED
        BBS     #RL_CS_V_NXM,UCB$W_DL_CS(R5),20$  ;IF SET - NONEXISTENT MEMORY
        BBC     #RL_CS_V_DE,UCB$W_DL_CS(R5),5$  ;IF CLR - NO DRIVE ERRORS
        BBC     #RL_MP_V_WL,UCB$W_DL_MP(R5),4$  ;IF CLR - NOT WRITE LOCKED
        BBS     #RL_MP_V_WGE,UCB$W_DL_MP(R5),20$  ;IF WL & WGE SET - WL ERROR
4$:     BITW    #RL_MP_M_WDE!-          ;WRITE DATA ERROR, OR
                RL_MP_M_CHE!-           ;CURRENT HEAD ERROR, OR
                RL_MP_M_WGE!-           ;WRITE GATE ERROR, OR
                RL_MP_M_DSE,UCB$W_DL_MP(R5)  ;DRIVE SELECT ERROR?
        BNEQ    20$                     ;IF NEQ - YES

;
; RETRIABLE ERROR EXIT
;
5$:     CVTBL   @UCB$L_DPC(R5),-(SP)    ;GET BRANCH DISPLACEMENT
        ADDL    (SP)+,UCB$L_DPC(R5)     ;CALCULATE RETURN ADDRESS - 1

;
; SUCCESSFUL OPERATION EXIT
;
10$:    INCL    UCB$L_DPC(R5)           ;ADJUST TO CORRECT RETURN ADDRESS
        JMP     @UCB$L_DPC(R5)          ;RETURN TO DRIVER

;
; FATAL ERROR EXIT
;
20$:    BRW     FATALERR                ;FATAL ERROR EXIT

;
; SPECIAL CONDITION EXIT (POWER FAILURE OR DEVICE TIMEOUT)
;
SPECOND:
        BBS     #UCB$V_POWER,UCB$W_STS(R5),PWRFAIL  ;IF SET - POWER FAILURE
                                        ;IF CLR - DEVICE TIMEOUT
        JSB     G^ERL$DEVICTMO          ;LOG DEVICE TIMEOUT
        BICW    #UCB$M_TIMOUT,UCB$W_STS(R5)  ;CLEAR TIMEOUT STATUS
        MOVZWL  #SS$_TIMEOUT,R0         ;SET DEVICE TIMEOUT STATUS
        DECB    UCB$B_ERTCNT(R5)        ;ANY ERROR RETRIES REMAINING?
        BEQL    RESETXFR                ;IF EQL - NO
        BRW     FDISPATCH               ;RETURN

RESETXFR:                               ;RESET TRANSFER BYTE COUNT
        MOVL    UCB$L_IRP(R5),R3        ;GET ADDRESS OF I/O PACKET
        MNEGW   IRP$W_BCNT(R3),UCB$W_BCR(R5)  ;RESET BYTE COUNT
        BRW     FUNCXT                  ;EXIT
```

```
PWRFAIL:                                ;POWER FAILURE
        BICW    #UCB$M_POWER,UCB$W_STS(R5)  ;CLEAR POWER FAILURE BIT
        TSTW    UCB$W_DL_DPN(R5)        ;ARE UCB RESOURCES ALLOCATED?
        BEQL    50$                     ;IF EQL - NO
        BBC     #UCB$V_DL_MAPPING,-     ;ADAPTER MAPPING?
                UCB$W_DL_FLAGS(R5),50$  ;IF BC NO
        RELDPR                          ;RELEASE DATA PATH
        RELMPR                          ;RELEASE MAP REGISTERS
50$:    RELCHAN                         ;RELEASE CHANNEL IF OWNED
        MOVL    UCB$L_IRP(R5),R3        ;GET ADDRESS OF I/O PACKET
        MOVQ    IRP$L_SVAPTE(R3),-      ;RESTORE TRANSFER PARAMETERS
                UCB$L_SVAPTE(R5)        ;...
        BRW     PREPROCESS              ;RETURN TO PREPROCESS UCB FIELDS
        .PAGE
        .SBTTL  INTERRUPT SERVICE ROUTINE
;++
; DL$INT - RL11 INTERRUPT SERVICE ROUTINE
;
; FUNCTIONAL DESCRIPTION:
;
;       THIS ROUTINE IS ENTERED VIA A JSB INSTRUCTION WHEN AN INTERRUPT
;       OCCURS ON AN RL11 DISK CONTROLLER. IF THE INTERRUPT IS NOT EXPECTED,
;       THE UNSOLICITED INTERRUPT ROUTINE DISMISSES THE INTERRUPT. IF
;       THE INTERRUPT IS EXPECTED, DEVICE REGISTERS ARE SAVED AND THE
;       DRIVER IS CALLED AT ITS INTERRUPT RETURN ADDRESS. THE DRIVER FORKS,
;       CAUSING A RETURN TO THIS ROUTINE, WHICH RESTORES GENERAL REGISTERS
;       AND DISMISSES THE INTERRUPT.
;
; INPUTS:
;
;       00(SP)  - POINTER TO ADDRESS OF THE IDB
;       04(SP)  - SAVED R0
;       08(SP)  - SAVED R1
;       12(SP)  - SAVED R2
;       16(SP)  - SAVED R3
;       20(SP)  - SAVED R4
;       24(SP)  - SAVED R5
;       28(SP)  - PC AT THE TIME OF THE INTERRUPT
;       32(SP)  - PSL AT THE TIME OF THE INTERRUPT
;
; OUTPUTS:
;
;       DEVICE REGISTERS ARE SAVED, IPL IS LOWERED TO FORK LEVEL, THE
;       INTERRUPT IS DISMISSED, ALL REGISTERS EXCEPT R0-R5 ARE PRESERVED.
;
;--
DL_INT::                                ;INTERRUPT SERVICE ROUTINE
        MOVL    @(SP)+,R3               ;REMOVE ADDRESS OF IDB FROM STACK
        ASSUME  IDB$L_CSR EQ 0
        ASSUME  IDB$L_OWNER EQ 4
        MOVQ    (R3),R4                 ;GET ADDRESS OF CSR AND UCB
        TSTL    R5                      ;IS R5 A ZERO
        BEQL    DL_UNSOLNT              ;IF EQL NO OWNER
        DEVICELOCK -
                LOCKADDR=UCB$L_DLCK(R5),- ;LOCK DEVICE ACCESS
                CONDITION=NOSETIPL,-    ;DON'T CHANGE IPL
                PRESERVE=NO             ;DON'T PRESERVE R0
        BBCC    #UCB$V_INT,-            ;IF CLR - INTERRUPT NOT EXPECTED
                UCB$W_STS(R5),40$       ;...
```

```
        CMPB    #CDF_READHEAD,UCB$B_CEX(R5)  ;READ HEADER FUNCTION?
        BNEQ    10$                          ;IF NEQ - NO
        MOVW    RL_MP(R4),UCB$W_DL_DB(R5)    ;SAVE SECTOR HEADER INFORMATION
        MOVW    RL_MP(R4),UCB$W_DL_DB+2(R5)  ;...
        MOVW    RL_MP(R4),UCB$W_DL_DB+4(R5)  ;...

10$:    MOVAB   RL_CS(R4),R2                 ;GET ADDRESS OF CONTROL STATUS REGISTER
        MOVAB   UCB$W_DL_CS(R5),R3           ;GET ADDRESS OF REGISTER SAVE AREA
        MOVW    (R2)+,(R3)+                  ;SAVE CONTROL STATUS REGISTER
        MOVW    (R2)+,(R3)+                  ;SAVE BUFFER ADDRESS REGISTER
        MOVW    (R2)+,(R3)+                  ;SAVE DISK ADDRESS REGISTER
        MOVW    (R2)+,(R3)+                  ;SAVE MULTIPURPOSE REGISTER

20$:    MOVQ    UCB$L_FR3(R5),R3             ;RESTORE DRIVER CONTEXT
        JSB     @UCB$L_FPC(R5)               ;CALL DRIVER AT INTERRUPT RETURN ADDRESS

40$:    DEVICEUNLOCK -
                LOCKADDR=UCB$L_DLCK(R5),-    ;UNLOCK DEVICE ACCESS
                PRESERVE=NO                  ;DON'T PRESERVE R0

DL_UNSOLNT:                                  ;UNSOLICITED INTERRUPT
        POPR    #^M<R0,R1,R2,R3,R4,R5>       ;RESTORE R0-R5
        REI                                  ;RETURN FROM INTERRUPT
        .PAGE

        .SBTTL  REGISTER DUMP ROUTINE
;++
;
; DL_REGDUMP - REGISTER DUMP ROUTINE
;
; FUNCTIONAL DESCRIPTION:
;
; THIS ROUTINE IS CALLED TO SAVE THE DEVICE REGISTERS AND UBA RESOURCE
; REGISTERS IN A SPECIFIED BUFFER. IT IS CALLED FROM THE DEVICE ERROR
; LOGGING ROUTINE AND FROM THE DIAGNOSTIC BUFFER FILL ROUTINE.
;
; INPUTS:
;
;       R0      - ADDRESS OF REGISTER SAVE BUFFER
;       R4      - ADDRESS OF DEVICE CONTROL STATUS REGISTER (CSR)
;       R5      - ADDRESS OF UNIT CONTROL BLOCK (UCB)
;
; OUTPUTS:
;
;       THE DEVICE AND UBA REGISTERS ARE SAVED IN THE SPECIFIED BUFFER.
;       R0 CONTAINS THE ADDRESS OF THE NEXT EMPTY LONGWORD IN THE BUFFER.
;       ALL REGISTERS EXCEPT R1 AND R2 ARE PRESERVED.
;
;--

DL_REGDUMP:                                  ;REGISTER DUMP ROUTINE
        MOVL    #<RL_NUM_REGS+5>,(R0)+       ;INSERT NUMBER OF REGISTERS
        MOVAL   UCB$W_DL_CS(R5),R1           ;GET ADDRESS OF SAVED DEVICE REGISTERS
        MOVZBL  #RL_NUM_REGS,R2              ;GET NUMBER OF DEVICE REGISTERS TO MOVE
10$:    MOVZWL  (R1)+,(R0)+                  ;DUMP REGISTER IN BUFFER
        SOBGTR  R2,10$                       ;IF GTR - STILL MORE TO MOVE
        MOVZWL  (R1)+,(R0)+                  ;DUMP DATAPATH NUMBER
        MOVL    (R1)+,(R0)+                  ;DUMP DATAPATH REGISTER
        MOVL    (R1)+,(R0)+                  ;DUMP FINAL MAP REGISTER
        MOVL    (R1)+,(R0)+                  ;DUMP PREVIOUS MAP REGISTER
        MOVZBL  (R1)+,(R0)+                  ;DUMP DATAPATH PURGE ERROR REGISTER
        RSB                                  ;RETURN
```

```
        .PAGE
        .SBTTL  MOVE TO USER BUFFER ROUTINE
;++
;
; DL_MOVE_TO_BUFFER - MOVE TO USER BUFFER
;
; FUNCTIONAL DESCRIPTION:
;
; THIS ROUTINE MOVES DATA BETWEEN THE PHYSICALLY CONTIGUOUS BUFFER AND
; THE USER'S BUFFER.
;
; INPUTS:
;
;       R5 - UCB ADDRESS
;
; OUTPUTS:
;
;       DATA MOVE BETWEEN THE PHYSICALLY CONTIGUOUS BUFFER AND THE USER'S BUFFER.
;       REGISTER'S R0,R1, AND R2 ARE DESTROYED
;
;--
DL_MOVE_TO_BUFFER:                      ;BUFFER MOVE ROUTINE
        BBS     #UCB$V_DL_MAPPING,-     ;ADAPTER MAPPING?
                UCB$W_DL_FLAGS(R5),10$  ;IF BS YES NOTHING TO MOVE
        CMPB    #CDF_READDATA,UCB$B_CEX(R5);READ DATA OPERATION?
        BNEQ    10$                     ;IF NEQ NOT A READ
        BBS     #0,UCB$B_DL_DCHEK(R5),- ;DATA CHECK IN PROGRESS?
                10$                     ;IF BS YES NOTHING TO MOVE
        TSTL    UCB$A_DL_MOVRTN(R5)     ;ANYTHING TO MOVE?
        BEQL    20$                     ;IF EQL NO
        MOVL    UCB$L_DL_BUFADR(R5),R0  ;GET USER BUFFER POINTER
        MOVL    UCB$A_DL_BUF_VA(R5),R1  ;GET PHYSICALLY CONTIGUOUS BUFFER ADDRESS
        MOVZWL  UCB$W_DL_PBCR(R5),R2    ;GET NUMBER OF BYTES TO TRANSFER
        JSB     @UCB$A_DL_MOVRTN(R5)    ;CALL MOVE ROUTINE
        MOVL    R0,UCB$L_DL_BUFADR(R5)  ;SAVE INTERNAL BUFFER POINTER
        MOVAB   G^IOC$MOVTOUSER2,-      ;SET NEXT MOVE ROUTINE TO BE USED
                UCB$A_DL_MOVRTN(R5)     ;
10$:    RSB                             ;RETURN
20$:    MOVAB   G^IOC$MOVTOUSER,-       ;SET NEXT MOVE ROUTINE TO BE USED
                UCB$A_DL_MOVRTN(R5)     ;
        RSB                             ;RETURN
```

```
        .PAGE
        .SBTTL  MOVE FROM USER BUFFER ROUTINE
;++
;
; DL_MOVE_FROM_BUFFER - MOVE FROM USER BUFFER
;
; FUNCTIONAL DESCRIPTION:
;
; THIS ROUTINE MOVES DATA BETWEEN THE PHYSICALLY CONTIGUOUS BUFFER AND
; THE USER'S BUFFER.
;
; INPUTS:
;
;       R5 - UCB ADDRESS
;
; OUTPUTS:
;
;       DATA MOVE BETWEEN THE PHYSICALLY CONTIGUOUS BUFFER AND THE USER'S BUFFER.
;       REGISTER'S R0,R1, AND R2 ARE DESTROYED
;
;--
DL_MOVE_FROM_BUFFER:                    ;BUFFER MOVE ROUTINE
        BBS     #UCB$V_DL_MAPPING,-     ;ADAPTER MAPPING?
                UCB$W_DL_FLAGS(R5),10$  ;IF BS YES NOTHING TO MOVE
        CMPB    #CDF_WRITEDATA,UCB$B_CEX(R5);WRITE DATA OPERATION?
        BNEQ    10$                     ;IF NEQ NOT A WRITE
        BBS     #0,UCB$B_DL_DCHEK(R5),- ;DATA CHECK IN PROGRESS?
                10$                     ;IF BS YES NOTHING TO MOVE
        MOVL    UCB$L_DL_BUFADR(R5),R0  ;GET USER BUFFER POINTER
        MOVL    UCB$A_DL_BUF_VA(R5),R1  ;GET PHYSICALLY CONTIGUOUS BUFFER ADDRESS
        MOVZWL  UCB$W_DL_PBCR(R5),R2    ;GET NUMBER OF BYTES TO TRANSFER
        JSB     @UCB$A_DL_MOVRTN(R5)    ;CALL MOVE ROUTINE
        MOVL    R0,UCB$L_DL_BUFADR(R5)  ;SAVE INTERNAL BUFFER POINTER
        MOVAB   G^IOC$MOVFRUSER2,-      ;SET NEXT MOVE ROUTINE TO BE USED
                UCB$A_DL_MOVRTN(R5)     ;
10$:    RSB                             ;RETURN

DL_END:                                 ;ADDRESS OF LAST LOCATION IN DRIVER
        .END
```

D Sample Driver for the DR11–W and DRV11–WA Interfaces

The following driver, XADRIVER, controls the DR11–W, a 16-bit parallel DMA interface on UNIBUS systems. The driver also controls the DRV11–WA, a 16-bit parallel DMA interface on the Q22 bus. Operational details of these devices, as well as the capabilities controlled by the driver, can be found in the *VMS I/O User's Reference Manual: Part II.*

You can find an online copy of the driver code (XADRIVER.MAR) in SYS$EXAMPLES.

```
        .TITLE  XADRIVER - VAX/VMS DR11-W AND DRV11-WA DRIVER
        .IDENT  'X-18'

;
;*****************************************************************************
;*                                                                           *
;*  COPYRIGHT (c) 1978, 1980, 1982, 1984, 1985, 1986 BY                      *
;*  DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASSACHUSETTS.                   *
;*  ALL RIGHTS RESERVED.                                                     *
;*                                                                           *
;*  THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED AND COPIED    *
;*  ONLY IN  ACCORDANCE WITH  THE  TERMS  OF  SUCH  LICENSE  AND WITH THE    *
;*  INCLUSION OF THE ABOVE COPYRIGHT NOTICE. THIS SOFTWARE OR  ANY  OTHER    *
;*  COPIES THEREOF MAY NOT BE PROVIDED OR OTHERWISE MADE AVAILABLE TO ANY    *
;*  OTHER PERSON.  NO TITLE TO AND OWNERSHIP OF  THE  SOFTWARE IS  HEREBY    *
;*  TRANSFERRED.                                                             *
;*                                                                           *
;*  THE INFORMATION IN THIS SOFTWARE IS  SUBJECT TO CHANGE WITHOUT NOTICE    *
;*  AND  SHOULD  NOT  BE  CONSTRUED AS  A COMMITMENT BY DIGITAL EQUIPMENT    *
;*  CORPORATION.                                                             *
;*                                                                           *
;*  DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE  OR  RELIABILITY OF ITS    *
;*  SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DIGITAL.                  *
;*                                                                           *
;*                                                                           *
;*****************************************************************************
;
;++
;
; FACILITY:
;
;       VAX/VMS Executive, I/O Drivers
;
; ABSTRACT:
;
;       This module contains the driver for the DR11-W (Unibus) and
;       DRV11-WA (Q-bus).  Since the driver was originally written for
;       the DR11-W, many inline comments refer to the "DR11-W" and "Unibus"
;       but apply equally well to the DRV11-WA and the Q-bus.
;
;               For DR11-W users:
;               This driver works for all hardware revision levels of
;               the DR11-W.
;
;               For DRV11-WA users:
;               This driver works for all hardware revision levels of
```

```
;               the DRV11-WA, up through and including CS Rev C.
;
;                  BECAUSE ETCH REVISION E OF THE DRV11-WA
;                  PROVIDES SEVERAL CUSTOMER MODIFIABLE SETTINGS,
;                  IT IS VERY IMPORTANT THAT THE USERS OF THIS
;                  BOARD PROPERLY CONFIGURE IT TO BE BACKWARDS
;                  COMPATIBLE WITH EARLIER REVISIONS OF THE
;                  DRV11-WA.  SPECIFICALLY, ON ETCH REVISION E
;                  BOARDS, JUMPERS W2, W3, AND W6 MUST BE INSTALLED.
;
;                                 ========
;
; ENVIRONMENT:
;
;       Kernel Mode, Non-paged
;
;--
        .SBTTL  External and local symbol definitions

; External symbols
        $ACBDEF                         ; AST control block
        $ADPDEF                         ; Adapter control block
        $CRBDEF                         ; Channel request block
        $DCDEF                          ; Device types
        $DDBDEF                         ; Device data block
        $DEVDEF                         ; Device characteristics
        $DPTDEF                         ; Driver prologue table
        $DYNDEF                         ; Dynamic data structure types
        $EMBDEF                         ; EMB offsets
        $IDBDEF                         ; Interrupt data block
        $IODEF                          ; I/O function codes
        $IPLDEF                         ; Hardware IPL definitions
        $IRPDEF                         ; I/O request packet
        $PRDEF                          ; Internal processor registers
        $PRIDEF                         ; Scheduler priority increments
        $SSDEF                          ; System status codes
        $UCBDEF                         ; Unit control block
        $VECDEF                         ; Interrupt vector block
        $XADEF                          ; Define device specific
                                        ; characteristics

; Local symbols

; Argument list (AP) offsets for device-dependent QIO parameters

P1      = 0                             ; First QIO parameter
P2      = 4                             ; Second QIO parameter
P3      = 8                             ; Third QIO parameter
P4      = 12                            ; Fourth QIO parameter
P5      = 16                            ; Fifth QIO parameter
P6      = 20                            ; Sixth QIO parameter

; Other constants

XA_DEF_TIMEOUT  = 10                    ; 10 second default device timeout
XA_DEF_BUFSIZ   = 65535                 ; Default buffer size
XA_RESET_DELAY  = <<2+9>/10>            ; Delay N microseconds after RESET
                                        ;(rounded up to 10 microsec intervals)

; DR11-W definitions that follow the standard UCB fields
; *** N O T E ***  ORDER OF THESE UCB FIELDS IS ASSUMED
```

```
        $DEFINI UCB
        .=UCB$L_DPC+4
$DEF    UCB$L_XA_ATTN                   ; Attention AST listhead
                .BLKL   1
$DEF    UCB$W_XA_CSRTMP                 ; Temporary storage of CSR image
                .BLKW   1
$DEF    UCB$W_XA_BARTMP                 ; Temporary storage of BAR image
                .BLKW   1
$DEF    UCB$W_XA_CSR                    ; Saved CSR on interrupt
                .BLKW   1
$DEF    UCB$W_XA_EIR                    ; Saved EIR on interrupt
                .BLKW   1
$DEF    UCB$W_XA_IDR                    ; Saved IDR on interrupt
                .BLKW   1
$DEF    UCB$W_XA_BAR                    ; Saved BAR register on interrupt
                .BLKW   1
$DEF    UCB$W_XA_WCR                    ; Saved WCR register on interrupt
                .BLKW   1
$DEF    UCB$W_XA_ERROR                  ; Saved device status flag
                .BLKW   1
$DEF    UCB$L_XA_DPR                    ; Data Path Register contents
                .BLKL   1
$DEF    UCB$L_XA_FMPR                   ; Final Map Register contents
                .BLKL   1
$DEF    UCB$L_XA_PMPR                   ; Previous Map Register contents
                .BLKL   1
$DEF    UCB$W_XA_DPRN                   ; Saved Datapath Register Number
                .BLKW   1               ; And Datapath Parity error flag
$DEF    UCB$W_XA_BAETMP                 ; Temporary storage of BAE (DRV11-WA
                .BLKW   1               ;   only)
$DEF    UCB$W_XA_BAE                    ; Saved BAE register (DRV11-WA only)
                .BLKW   1

; Bit positions for device-dependent status field in UCB

        $VIELD  UCB,0,<-                ; UCB device specific bit definitions
                <ATTNAST,,M>,-          ; ATTN AST requested
                <UNEXPT,,M>,-           ; Unexpected interrupt received
                <IGNORE_UNEXPT,,M>,-    ; Ignore initial interrupt on DRV11-WA
                >
UCB$K_SIZE=.
        $DEFEND UCB

; Device register offsets from CSR address

        $DEFINI XA                      ; Start of DR11-W definitions
$DEF    XA_WCR                          ; Word count
                        .BLKW   1
$DEF    XA_BAR                          ; Buffer address
$DEF    XA_BAE                          ; Buffer address extension (DRV11-WA)
                        .BLKW   1
$DEF    XA_CSR                          ; Control/status

; Bit positions for device control/status register

        $EQULST XA$K_,,0,1,<-           ; Define CSR FNCT bit values
                <FNCT1,2>-
                <FNCT2,4>-
                <FNCT3,8>-
                <STATUSA,2048>-         ; Define CSR STATUS bit values
                <STATUSB,1024>-
                <STATUSC,512>-
        >
```

```
        $VIELD  XA_CSR,0,<-             ; Control/status register
                <GO,,M>,-               ; Start device
                <FNCT,3,M>,-            ; CSR FNCT bits
                <XBA,2,M>,-             ; Extended address bits
                <IE,,M>,-               ; Enable interrupts
                <RDY,,M>,-              ; Device ready for command
                <CYCLE,,M>,-            ; Starts slave transmit
                <STATUS,3,M>,-          ; CSR STATUS bits
                <MAINT,,M>,-            ; Maintenance bit
                <ATTN,,M>,-             ; Status from other processor
                <NEX,,M>,-              ; Nonexistent memory flag
                <ERROR,,M>,-            ; Error or external interrupt
        >

        $VIELD  XA_BAE,0,<-             ; Extended bus address register
                <MSB_ADDR,6,M>,-        ; Qbus physical address <22:16>
                <,9,>,-
                <CS_REV_C,,M>,-         ; true if DRV11-WA CS Rev C
        >

$DEF    XA_EIR                          ; Error information register

; Bit positions for error information register

        $VIELD  XA_EIR,0,<-             ; Error information register
                <REGFLG,,M>,-           ;Flags whether EIR or CSR is accessed
                <SPARE,7,M>,-           ; Unused - spare
                <BURST,,M>,-            ; Burst mode transfer occurred
                <DLT,,M>,-              ; Timeout for successive burst xfer
                <PAR,,M>,-              ; Parity error during DATI/P
                <ACLO,,M>,-             ; Power fail on this processor
                <MULTI,,M>,-            ; Multi-cycle request error
                <ATTN,,M>,-             ; ATTN - same as in CSR
                <NEX,,M>,-              ; NEX - same as in CSR
                <ERROR,,M>,-            ; ERROR - same as in CSR
        >
                .BLKW   1

$DEF    XA_IDR                          ; Input Data Buffer register
$DEF    XA_ODR                          ; Output Data Buffer register
                .BLKW   1

        $DEFEND XA                      ; End of DR11-W definitions


        .SBTTL  Device Driver Tables

; Driver prologue table
```

```
        DPTAB   -                                       ; DPT-creation macro
                END=XA_END,-                            ; End of driver label
                ADAPTER=UBA,-                           ; Adapter type
                FLAGS=DPT$M_SVP,-                       ; Allocate system page table
                UCBSIZE=UCB$K_SIZE,-                    ; UCB size
                NAME=XADRIVER                           ; Driver name
        DPT_STORE INIT                                  ; Start of load
                                                        ; initialization table
        DPT_STORE UCB,UCB$B_FLCK,B,SPL$C_IOLOCK8 ; Device fork IPL
        DPT_STORE UCB,UCB$B_DIPL,B,22                   ; Device interrupt IPL
        DPT_STORE UCB,UCB$L_DEVCHAR,L,<-                ; Device characteristics
                DEV$M_AVL!-                             ; Available
                DEV$M_RTM!-                             ; Real Time device
                DEV$M_ELG!-                             ; Error Logging enabled
                DEV$M_IDV!-                             ;   input device
                DEV$M_ODV>                              ;   output device
        DPT_STORE UCB,UCB$B_DEVCLASS,B,DC$_REALTIME     ; Device class
        DPT_STORE UCB,UCB$B_DEVTYPE,B,DT$_DR11W ; Device Type
        DPT_STORE UCB,UCB$W_DEVBUFSIZ,W,-               ; Default buffer size
                XA_DEF_BUFSIZ
        DPT_STORE REINIT                                ; Start of reload
                                                        ; initialization table
        DPT_STORE DDB,DDB$L_DDT,D,XA$DDT                ; Address of DDT
        DPT_STORE CRB,CRB$L_INTD+4,D,-                  ; Address of interrupt
                XA_INTERRUPT                            ; service routine
        DPT_STORE CRB,CRB$L_INTD+VEC$L_INITIAL,-; Address of controller
                D,XA_CONTROL_INIT                       ; initialization routine
        DPT_STORE END                                   ; End of initialization
                                                        ; tables

; Driver dispatch table

        DDTAB   -                                       ; DDT-creation macro
                DEVNAM=XA,-                             ; Name of device
                START=XA_START,-                        ; Start I/O routine
                FUNCTB=XA_FUNCTABLE,-                   ; FDT address
                CANCEL=XA_CANCEL,-                      ; Cancel I/O routine
                REGDMP=XA_REGDUMP,-                     ; Register dump routine
                DIAGBF=<<15*4>+<<3+5+1>*4>>,-   ; Diagnostic buffer size
                ERLGBF=<<15*4>+<1*4>+<EMB$L_DV_REGSAV>> ;Error log buffer size

;
; Function dispatch table
;
XA_FUNCTABLE:                                           ; FDT for driver
        FUNCTAB ,-                                      ; Valid I/O functions
                <READPBLK,READLBLK,READVBLK,WRITEPBLK,WRITELBLK,WRITEVBLK,-
                SETMODE,SETCHAR,SENSEMODE,SENSECHAR>
        FUNCTAB ,                                       ; No buffered functions
        FUNCTAB XA_READ_WRITE,-                         ; Device-specific FDT
                <READPBLK,READLBLK,READVBLK,WRITEPBLK,WRITELBLK,WRITEVBLK>
        FUNCTAB +EXE$READ,<READPBLK,READLBLK,READVBLK>
        FUNCTAB +EXE$WRITE,<WRITEPBLK,WRITELBLK,WRITEVBLK>
        FUNCTAB XA_SETMODE,<SETMODE,SETCHAR>
        FUNCTAB +EXE$SENSEMODE,<SENSEMODE,SENSECHAR>

        .SBTTL  XA_CONTROL_INIT, Controller initialization
```

```
;++
; XA_CONTROL_INIT, Called when driver is loaded, system is booted, or
; power failure recovery.
;
; Functional Description:
;
;       1) Allocates the direct data path permanently
;       2) Assigns the controller data channel permanently
;       3) Clears the Control and Status Register
;       4) If power recovery, requests device timeout
;
; Inputs:
;
;       R4 = address of CSR
;       R5 = address of IDB
;       R6 = address of DDB
;       R8 = address of CRB
;
; Outputs:
;
;       VEC$V_PATHLOCK bit set in CRB$L_INTD+VEC$B_DATAPATH
;       UCB address placed into IDB$L_OWNER
;
;
;--
XA_CONTROL_INIT:
        MOVL    IDB$L_UCBLST(R5),R0     ; Address of UCB
        MOVL    R0,IDB$L_OWNER(R5)      ; Make permanent controller owner
        BISW    #UCB$M_ONLINE,UCB$W_STS(R0)
                                        ; Set device status "on-line"
        ADPDISP SELECT=ADAP_MAPPING,-   ; Check for adapter mapping
                ADDRLIST=<<YES,1$>>,-
                CRBADDR=R8,-
                SCRATCH=R1
        BUG_CHECK UNSUPRTCPU,FATAL      ; DRV11-WA not supported on non-
                                        ; mapping adapter
1$:     ADPDISP SELECT=QBUS,-           ; Check for QBUS machine
                ADDRLIST=<<NO,9$>>,-
                ADPADDR=R1
        BLBC    g^SGN$GB_QBUS_MULT_INTR,2$ ; protect against ILLQBUSCFG
        MOVB    #^x14,UCB$B_DIPL(R5)    ; This is correct DIPL for DRV11-WA
2$:     MOVB    #DT$_XA_DRV11WA,-       ; This is a Q-bus, therefore it is
                  UCB$B_DEVTYPE(R0)     ;   a DRV11-WA rather than a DR11-W.

;+
;
;  DRV11-WAs at CS revision B and earlier incorrectly generated an interrupt
;  whenever the Interrupt Enable control bit (IE) underwent a low to high
;  transition.  This phenomenon does not occur in boards at CS revision C.
;
;  To account for this unsoliciated interrupt, IGNORE_UNEXPT is set at
;  initialization for all DRV11-WAs at CS revisions prior to C.  When this
;  bit is set, the next unexpected interrupt (as determined by the INT bit
;  in UCB status word, which is set whenever an I/O request is outstanding)
;  is discarded.  The IGNORE_UNEXPT flag is necessary because driver
;  initialization occurs at a different IPL than the interrupt handling
;  routine.
;
;-

        BICW    #UCB$M_IGNORE_UNEXPT,-  ; start out assuming this is a
                  UCB$W_DEVSTS(R0)      ; CS Rev C DRV11-WA
```

```
        TSTW    XA_BAR(R4)              ; BAR and BAE share the same physical
        MOVW    XA_BAE(R4),R1           ; address- they must be read in order

        BBS     #XA_BAE$V_CS_REV_C,R1,9$        ; branch around if CS Rev C

        ;  BAE<15> is always set if the DRV11-WA is at CS Rev C or later.

        BISW    #UCB$M_IGNORE_UNEXPT,-  ; set flag so all interrupts are
                   UCB$W_DEVSTS(R0)     ; discarded until further notice

; If powerfail has occurred and device was active, force device timeout.
; The user can set his own timeout interval for each request.  Time-
; out is forced so a very long timeout period will be short circuited.

9$:     BBS     #UCB$V_POWER,UCB$W_STS(R0),10$
                                        ; Branch if powerfail
        BISB    #VEC$M_PATHLOCK,CRB$L_INTD+VEC$B_DATAPATH(R8)
                                        ; Permanently allocate direct datapath
10$:    PUSHL   R5                      ; Save R5
        MOVL    R0,R5                   ; Copy UCB address to R5
        BSBW    XA_DEV_HWRESET
        POPL    R5                      ; Restore R5
        RSB                             ; Done

        .SBTTL  XA_READ_WRITE, FDT for device data transfers

;++
; XA_READ_WRITE, FDT for READLBLK,READVBLK,READPBLK,WRITELBLK,WRITEVBLK,
;                        WRITEPBLK
;
; Functional description:
;
;       1) Rejects QUEUE I/O's with odd transfer count
;       2) Rejects QUEUE I/O's for BLOCK MODE request to UBA Direct Data
;          PATH on odd byte boundary
;       3) Stores request timeout count specified in P3 into IRP
;       4) Stores FNCT bits specified in P4 into IRP
;       5) Stores word to write into ODR from P5 into IRP
;       6) Checks block mode transfers for memory modify access
;
; Inputs:
;
;       R3 = Address of IRP
;       R4 = Address of PCB
;       R5 = Address of UCB
;       R6 = Address of CCB
;       R8 = Address of FDT routine
;       AP = Address of P1
;               P1 = Buffer Address
;               P2 = Buffer size in bytes
;               P3 = Request timeout period (conditional on IO$M_TIMED)
;               P4 = Value for CSR FNCT bits (conditional on IO$M_SETFNCT)
;               P5 = Value for ODR (conditional on IO$M_SETFNCT)
;               P6 = Address of Diagnostic Buffer
;
; Outputs:
;
;       R0 = Error status if odd transfer count
;       IRP$L_MEDIA = Timeout count for this request
;       IRP$L_SEGVBN = FNCT bits for DR11-W CSR and ODR image
;
;--

XA_READ_WRITE:
```

```
; The IO$M_INHERLOG ("inhibit error logging") function modifier was not
; intended to be used by this driver.  However, since the definition for
; the IO$M_RESET modifier used to be the same as that for IO$M_INHERLOG,
; the error logging routine incorrectly used the IO$M_RESET bit to
; determine whether it should log errors.  To solve this problem, the
; definition for IO$M_RESET was changed.  For the sake of old programs, we
; manually move the RESET bit to its new location.

        BBCC    #IO$V_INHERLOG,IRP$W_FUNC(R3),1$
                                        ; Branch if old reset bit not set
        BISW    #IO$M_RESET,IRP$W_FUNC(R3)
                                        ; Set new reset bit
1$:     BLBC    P2(AP),10$              ; Branch if transfer count even
2$:     MOVZWL  #SS$_BADPARAM,R0        ; Set error status code
5$:     JMP     G^EXE$ABORTIO           ; Abort request
10$:    MOVZWL  IRP$W_FUNC(R3),R1       ; Fetch I/O Function code
        MOVL    P3(AP),IRP$L_MEDIA(R3)  ; Set request specific timeout count
        BBS     #IO$V_TIMED,R1,15$      ; Branch if timeout specified
        MOVL    #XA_DEF_TIMEOUT,IRP$L_MEDIA(R3)
                                        ; Else set default timeout value
15$:    BBC     #IO$V_DIAGNOSTIC,R1,20$ ; Branch if not maintenance reqeust
        EXTZV   #IO$V_FCODE,#IO$S_FCODE,R1,R1 ; AND out all function modifiers
        CMPB    #IO$_READPBLK,R1        ; If maintenance function, must be
                                        ; physical I/O read or write
        BEQL    20$
        CMPB    #IO$_WRITEPBLK,R1
        BEQL    20$
        MOVZWL  #SS$_NOPRIV,R0          ; No privilege for operation
        BRB     5$                      ; Abort request
20$:    EXTZV   #0,#3,P4(AP),R0         ; Get value for FNCT bits
        ASHL    #XA_CSR$V_FNCT,R0,IRP$L_SEGVBN(R3) ; Shift into position
                                                   ; for CSR
        MOVW    P5(AP),IRP$L_SEGVBN+2(R3) ; Store ODR value for later

; If this is a block mode transfer, check buffer for modify access
; whether or not the function is read or write.  The DR11-W does
; not decide whether to read or write, the user's device does.
; For word mode requests, return to read check or write check.
;
; If this is a BLOCK MODE request and the UBA Direct Data Path is
; in use, check the data buffer address for word alignment.  If buffer
; is not word aligned, reject the request.

        BBS     #IO$V_WORD,IRP$W_FUNC(R3),30$
                                        ; Branch if word mode transfer
        BBS     #XA$V_DATAPATH,UCB$L_DEVDEPEND(R5),25$
                                        ; Branch if Buffered Data Path in use
        BLBS    P1(AP),2$               ; DDP, branch on bad alignment
25$:    JMP     G^EXE$MODIFY            ; Check buffer for modify access
30$:    RSB                             ; Return

        .SBTTL  XA_SETMODE, Set Mode, Set characteristics FDT
```

```
;++
; XA_SETMODE, FDT routine to process SET MODE and SET CHARACTERISTICS
;
; Functional description:
;
;       If IO$M_ATTNAST modifier is set, queue attention AST for device
;       If IO$M_DATAPATH modifier is set, queue packet.
;       Else, finish I/O.
;
; Inputs:
;
;       R3 = I/O packet address
;       R4 = PCB address
;       R5 = UCB address
;       R6 = CCB address
;       R7 = Function code
;       AP = QIO Parameter list address
;
; Outputs:
;
;       If IO$M_ATTNAST is specified, queue AST on UCB attention AST list.
;       If IO$M_DATAPATH is specified, queue packet to driver.
;       Else, use exec routine to update device characteristics
;
;--
XA_SETMODE:
        MOVZWL  IRP$W_FUNC(R3),R0       ; Get entire function code
        BBC     #IO$V_ATTNAST,R0,20$    ; Branch if not an attention AST

; Attention AST request

        PUSHR   #^M<R4,R7>
        MOVAB   UCB$L_XA_ATTN(R5),R7    ; Address of attention AST control
                                        ; block list
        JSB     G^COM$SETATTNAST        ; Set up attention AST
        POPR    #^M<R4,R7>
        BLBC    R0,50$                  ; Branch if error
        BISW    #UCB$M_ATTNAST,UCB$W_DEVSTS(R5)
                                        ; Flag ATTN AST expected.
        BBC     #UCB$V_UNEXPT,UCB$W_DEVSTS(R5),10$
                                        ; Deliver AST if unsolicited interrupt
        BSBW    DEL_ATTNAST
10$:    MOVZBL  #SS$_NORMAL,R0          ; Set status
        JMP     G^EXE$FINISHIOC         ; That's all for now (clears R1)

; If modifier IO$M_DATAPATH is set,
; queue packet.  The data path is changed at driver level to preserve
; order with other requests.

20$:    BBS     S^#IO$V_DATAPATH,R0,30$ ; If BDP modifier set, queue packet

        JMP     G^EXE$SETCHAR           ; Set device characteristics

; This is a request to change data path useage, queue packet

30$:    CMPL    #IO$_SETCHAR,R7         ; Set characteristics?
        BNEQ    45$                     ; No, must have the privilege
        JMP     G^EXE$SETMODE           ; Queue packet to start I/O

; Error, abort IO

45$:    MOVZWL  #SS$_NOPRIV,R0          ; No priv for operation
50$:    CLRL    R1
        JMP     G^EXE$ABORTIO           ; Abort IO on error
```

```
        .SBTTL  XA_START, Start I/O routines
;++
; XA_START - Start a data transfer, set characteristics, enable ATTN AST.
;
; Functional Description:
;
;       This routine has two major functions:
;
;       1) Start an I/O transfer.  This transfer can be in either word
;          or block mode.  The FNCTN bits in the DR11-W CSR are set.  If
;          the transfer count is zero, the STATUS bits in the DR11-W CSR
;          are read and the request completed.
;       2) Set Characteristics.  If the function is change data path, the
;          new data path flag is set in the UCB.
;
; Inputs:
;
;       R3 = Address of the I/O request packet
;       R5 = Address of the UCB
;
; Outputs:
;
;       R0 = final status and number of bytes transferred
;       R1 = value of CSR STATUS bits and value of input data buffer register
;       Device errors are logged
;       Diagnostic buffer is filled
;
;--
        .ENABL  LSB

XA_START:

; Retrieve the address of the device CSR

        ASSUME  IDB$L_CSR EQ 0
        MOVL    UCB$L_CRB(R5),R4        ; Address of CRB
        MOVL    @CRB$L_INTD+VEC$L_IDB(R4),R4
                                        ; Address of CSR

; Fetch the I/O function code

        MOVZWL  IRP$W_FUNC(R3),R1       ; Get entire function code
        MOVW    R1,UCB$W_FUNC(R5)       ; Save FUNC in UCB for Error Logging
        EXTZV   #IO$V_FCODE,#IO$S_FCODE,R1,R2 ; Extract function field

; Dispatch on function code.  If this is SET CHARACTERISTICS, we will
; select a data path for future use.
; If this is a transfer function, it will either be processed in word
; or block mode.

        CMPB    #IO$_SETCHAR,R2         ; Set characteristics?
        BNEQ    3$
```

```
;++
; SET CHARACTERISTICS - Process Set Characteristics QIO function
;
; INPUTS:
;
;       XA_DATAPATH bit in Device Characteristics specifies which data path
;       to use.  If bit is a one, use buffered data path.  If zero, use
;       direct datapath.
;
; OUTPUTS:
;
;       CRB is flagged as to which datapath to use.
;       DEVDEPEND bits in device characteristics is updated
;               XA_DATAPATH = 1 -> buffered data path in use
;               XA_DATAPATH = 0 -> direct data path in use
;--

        MOVL    UCB$L_CRB(R5),R0                ; Get CRB address
        MOVQ    IRP$L_MEDIA(R3),UCB$B_DEVCLASS(R5) ;Set device characteristics
        BISB    #VEC$M_PATHLOCK,CRB$L_INTD+VEC$B_DATAPATH(R0)
                                                ; Assume direct datapath
        BBC     #XA$V_DATAPATH,UCB$L_DEVDEPEND(R5),2$ ;  Were we right?
        BICB    #VEC$M_PATHLOCK,CRB$L_INTD+VEC$B_DATAPATH(R0) ;Set buffered
                                                              ;datapath
2$:
        CLRL    R1                              ; Return Success
        MOVZWL  #SS$_NORMAL,R0
        REQCOM

; If subfunction modifier for device reset is set, do one here

3$:     BBC     S^#IO$V_RESET,R1,4$     ; Branch if not device reset
        BSBW    XA_DEV_RESET            ; Reset DR11-W

; This must be a data transfer function - i.e. READ OR WRITE
; Check to see if this is a zero length transfer.
; If so, only set CSR FNCT bits and return STATUS from CSR

4$:     TSTW    UCB$W_BCNT(R5)          ; Is transfer count zero?
        BNEQ    10$                     ; No, continue with data transfer
        BBC     S^#IO$V_SETFNCT,R1,6$   ; Set CSR FNCT specified?
        DEVICELOCK -
                LOCKADDR=UCB$L_DLCK(R5),- ; Lock device access
                SAVIPL=-(SP),-          ; Save current IPL
                PRESERVE=NO             ; Don't preserve R0
        MOVW    IRP$L_SEGVBN+2(R3),XA_ODR(R4)
                                        ; Store word in ODR
        MOVZWL  XA_CSR(R4),R0
        BICW    #<XA_CSR$M_FNCT!XA_CSR$M_ERROR>,R0
        BISW    IRP$L_SEGVBN(R3),R0
        BISW    #XA_CSR$M_ATTN,R0       ; Force ATTN on to prevent lost
                                        ; interrupt
        MOVW    R0,XA_CSR(R4)
        BBC     #XA$V_LINK,UCB$L_DEVDEPEND(R5),5$ ; Link mode?
        BICW3   #XA$K_FNCT2,R0,XA_CSR(R4)       ; Make FNCT bit 2 a pulse
5$:
        DEVICEUNLOCK -
                LOCKADDR=UCB$L_DLCK(R5),- ; Unlock device access
                NEWIPL=(SP)+,-          ; Enable interrupts
                PRESERVE=NO
6$:
        BSBW    XA_REGISTER             ; Fetch DR11-W registers
        BLBS    R0,7$                   ; If error, then log it
        JSB     G^ERL$DEVICERR          ; Log a device error
7$:     JSB     G^IOC$DIAGBUFILL        ; Fill diagnostic buffer if specified
```

```
        MOVL    UCB$W_XA_CSR(R5),R1     ; Return CSR and EIR in R1
        MOVZWL  UCB$W_XA_ERROR(R5),R0   ; Return status in R0
        BISB    #XA_CSR$M_IE,XA_CSR(R4) ; Enable device interrupts
        REQCOM                          ; Request done

; Build CSR image in R0 for later use in starting transfers

10$:
        MOVZWL  UCB$W_BCNT(R5),R0       ; Fetch byte count
        DIVL3   #2,R0,UCB$L_XA_DPR(R5)  ; Make byte count into word count
        ;
        ; Set up UCB$W_CSRTMP used for loading CSR later
        ;
        MOVZWL  XA_CSR(R4),R0
        BICW    #^C<XA_CSR$M_FNCT>,R0
        BISW    #XA_CSR$M_IE!XA_CSR$M_ATTN,R0   ; Set Interrupt Enable and ATTN
        BBC     S^#IO$V_SETFNCT,R1,20$  ; Set FNCT bits in CSR?
        BICW    #<XA_CSR$M_FNCT>,R0     ; Yes, Clear previous FNCT bits
        BISB    IRP$L_SEGVBN(R3),R0     ; OR in new value
20$:    BBC     S^#IO$V_DIAGNOSTIC,R1,23$  ; Check for maintenance function
        BISW    #XA_CSR$M_MAINT,R0      ; Set maintenance bit in CSR image

; Is this a word mode or block mode request?

23$:    MOVW    R0,UCB$W_XA_CSRTMP(R5)  ; Save CSR image in UCB
        BBC     S^#IO$V_WORD,R1,BLOCK_MODE ; Check if word or block mode
        BRW     WORD_MODE               ; Branch to handle word mode


;++
; BLOCK MODE -- Process a Block Mode (DMA) transfer request
;
; FUNCTIONAL DESCRIPTION:
;
;       This routine takes the buffer address, buffer size, function code,
;       and function modifier fields from the IRP.  It calculates the UNIBUS
;       address, allocates the UBA map registers, loads the DR11-W device
;       registers and starts the request.
;--
; Set up UBA
; Start transfer

BLOCK_MODE:

; If IO$M_CYCLE subfunction is specified, set CYCLE bit in CSR image

        BBC     #IO$V_CYCLE,R1,25$      ; Set CYCLE bit in CSR?
        BISW    #XA_CSR$M_CYCLE,UCB$W_XA_CSRTMP(R5) ; If yes, OR into CSR image

; Allocate UBA data path and map registers

25$:
        REQDPR                          ; Request UBA data path
        REQMPR                          ; Request UBA map registers
        LOADUBA                         ; Load UBA map reqisters

; Calculate the UNIBUS transfer address for the DR11-W from the UBA
; map register address and byte offset.
```

```
        MOVZWL  UCB$W_BOFF(R5),R1       ; Byte offset in first page of xfer
        MOVL    UCB$L_CRB(R5),R2        ; Address of CRB
        INSV    CRB$L_INTD+VEC$W_MAPREG(R2),#9,#9,R1
                                        ; Insert page number
        EXTZV   #16,#2,R1,R2            ; Extract bits 17:16 of bus address
        CMPB    #DT$_DR11W,-            ; If this is a DR11-W,
                  UCB$B_DEVTYPE(R5)
        BEQL    100$                    ;   then branch.
        MOVW    R2,UCB$W_XA_BAETMP(R5)  ; Save value of BAE prior to transfer
        CLRL    R2                      ; Clear XBA bits
100$:   ASHL    #XA_CSR$V_XBA,R2,R2     ; Shift extended memory bits for CSR
        BISW    #XA_CSR$M_GO,R2         ; Set "GO" bit into CSR image
        BISW    R2,UCB$W_XA_CSRTMP(R5)  ; Set into CSR image we are building
        BICW3   #<XA_CSR$M_GO!XA_CSR$M_CYCLE>,UCB$W_XA_CSRTMP(R5),R0
                                        ; CSR image less "GO" and "CYCLE"
        BICW3   #XA$K_FNCT2,UCB$W_XA_CSRTMP(R5),R2 ; CSR image less FNCT bit 2
        MOVW    R1,UCB$W_XA_BARTMP(R5)  ; Save BAR for error logging

; At this juncture:
;       R0 = CSR image less "GO" and "CYCLE"
;       R1 = low 16 bits of transfer bus address
;       R2 = CSR image less FNCT bit 2
;       UCB$L_XA_DPR(R5) = transfer count in words
;       UCB$W_XA_CSRTMP(R5) = CSR image to start transfer with

; Set DR11-W registers and start transfer
; Note that read-modify-write cycles are NOT performed to the DR11-W CSR.
; The CSR is always written directly into.  This prevents inadvertently
; setting the EIR select flag (writing bit 15) if error happens to become
; true.

        DEVICELOCK -
                LOCKADDR=UCB$L_DLCK(R5),- ; Lock device access
                SAVIPL=-(SP),-          ; Save current IPL
                PRESERVE=YES            ; Preserve R0
        SETIPL  #31,-                   ; Raise to IPL$_POWER
                ENVIRON=UNIPROCESSOR
        MNEGW   UCB$L_XA_DPR(R5),XA_WCR(R4)
                                        ; Load negative of transfer count
        MOVW    R1,XA_BAR(R4)           ; Load low 16 bits of bus address
        CMPB    #DT$_DR11W,-            ; If this is a DR11-W,
                  UCB$B_DEVTYPE(R5)
        BEQL    200$                    ;   then branch.
        MOVW    UCB$W_XA_BAETMP(R5),-   ; Load high bits of bus address
                  XA_BAE(R4)
200$:   MOVW    R0,XA_CSR(R4)          ;Load CSR image less "GO" and "CYCLE"
        BBC     #XA$V_LINK,UCB$L_DEVDEPEND(R5),26$ ;Link mode?
        MOVW    R2,XA_CSR(R4)   ;Yes, load CSR image less "FNCT" bit 2
        BRB     126$            ;Only if link mode in dev characteristics
26$:
        MOVW    UCB$W_XA_CSRTMP(R5),XA_CSR(R4) ; Move all bits to CSR

; Wait for transfer complete interrupt, powerfail, or device timeout

126$:
        WFIKPCH XA_TIME_OUT,IRP$L_MEDIA(R3) ; Wait for interrupt

; Device has interrupted, FORK

        IOFORK                          ; FORK to lower IPL

; Handle request completion, release UBA resources, check for errors
```

```
        MOVZWL  #SS$_NORMAL,-(SP)       ; Assume success, store code on stack
        CLRW    UCB$W_XA_DPRN(R5)       ; Clear DPR number and DPR error flag
        PURDPR                          ; Purge UBA buffered data path
        BLBS    R0,27$                  ; Branch if no datapath error
        MOVZWL  #SS$_PARITY,(SP)        ; Flag parity error on device
        INCB    UCB$W_XA_DPRN+1(R5)     ; Flag PDR error for log
27$:    MOVL    R1,UCB$L_XA_DPR(R5)     ; Save data path register in UCB
        EXTZV   #VEC$V_DATAPATH,-       ; Get Datapath register no.
                #VEC$S_DATAPATH,-       ; For Error Log
                CRB$L_INTD+VEC$B_DATAPATH(R3),R0
        MOVB    R0,UCB$W_XA_DPRN(R5)    ; Save for later in UCB
        EXTZV   #9,#7,UCB$W_XA_BAR(R5),R0 ; Low bits, final map register no.
        CMPB    #DT$_DR11W,-            ; If this is a DR11-W,
                  UCB$B_DEVTYPE(R5)
        BEQL    300$                    ;   then branch.
        MOVZWL  UCB$W_XA_BAE(R5),R1     ; Fetch high bits of map register no.
        BRB     310$
300$:   EXTZV   #4,#2,UCB$W_XA_CSR(R5),R1 ; Hi bits of map register no.
310$:   INSV    R1,#7,#2,R0             ; Entire map register number
        CMPW    R0,#496                 ; Is map register number in range?
        BGTR    28$                     ; No, forget it - compound error
        MOVL    (R2)[R0],UCB$L_XA_FMPR(R5) ; Save map register contents
        CLRL    UCB$L_XA_PMPR(R5)       ; Assume no previous map register
        DECL    R0                      ; Was there a previous map register?
        CMPV    #VEC$V_MAPREG,#VEC$S_MAPREG,-
                CRB$L_INTD+VEC$W_MAPREG(R3),R0
        BGTR    28$                     ; No if gtr
        MOVL    (R2)[R0],UCB$L_XA_PMPR(R5) ; Save previous map register
                                        ;  contents
28$:    RELMPR                          ; Release UBA resources
        RELDPR

; Check for errors and return status

        TSTW    UCB$W_XA_WCR(R5)        ; All words transferred?
        BEQL    30$                     ; Yes
        MOVZWL  #SS$_OPINCOMPL,(SP)     ; No, flag operation not complete
30$:    BBC     #XA_CSR$V_ERROR,UCB$W_XA_CSR(R5),35$ ; Branch on CSR error bit
        MOVZWL  UCB$W_XA_ERROR(R5),(SP) ; Flag for controller/drive error
                                        ; status
        BSBW    XA_DEV_RESET            ; Reset DR11-W
35$:    BLBS    (SP),40$                ; Any errors after all this?

        CMPW    (SP),#SS$_OPINCOMPL     ; Log the error, unless this is
        BNEQ    37$                     ;   a DRV11-WA running in link mode
        CMPB    #DT$_DR11W,-            ;   and the operation is incomplete,
                  UCB$B_DEVTYPE(R5)     ;   in which case it is an expected
        BEQL    37$                     ;   error and not worth logging.
        BBS     #XA$V_LINK,-            ;   ...
                  UCB$L_DEVDEPEND(R5),40$ ; ...
37$:    JSB     G^ERL$DEVICERR          ; Log the error.

40$:    BSBW    DEL_ATTNAST             ; Deliver outstanding ATTN ASTs
        JSB     G^IOC$DIAGBUFILL        ; Fill diagnostic buffer
        MOVL    (SP)+,R0                ; Get final device status
        MULW3   #2,UCB$W_XA_WCR(R5),R1  ; Calculate final transfer count
        ADDW    UCB$W_BCNT(R5),R1
        INSV    R1,#16,#16,R0           ; Insert into high byte of IOSB
        MOVL    UCB$W_XA_CSR(R5),R1     ; Return CSR and EIR in IOSB
        BISB    #XA_CSR$M_IE,XA_CSR(R4) ; Enable interrupts
        REQCOM                          ; Finish request in exec
```

```
        .DSABL  LSB
;++
; WORD MODE -- Process word mode (interrupt per word) transfer
;
; FUNCTIONAL DESCRIPTION:
;
;   Data is transferred one word at a time with an interrupt for each word.
;   The request is handled separately for a write (from memory to DR11-W
;   and a read (from DR11-W to memory).
;   For a write, data is fetched from memory, loaded into the ODR of the
;   DR11-W and the system waits for an interrupt.  For a read, the system
;   waits for a DR11-W interrupt and the IDR is transferred into memory.
;   If the unsolicited interrupt flag is set, the first word is transferred
;   directly into memory withou waiting for an interrupt.
;--
        .ENABL  LSB
WORD_MODE:

; Dispatch to separate loops on READ or WRITE

        CMPB    #IO$_READPBLK,R2        ; Check for read function
        BNEQ    10$                     ; Br if not, must be write function
        BRW     30$                     ; Else, read

;++
; WORD MODE WRITE -- Write (output) in word mode
;
; FUNCTIONAL DESCRIPTION:
;
;       Transfer the requested number of words from user memory to
;       the DR11-W ODR one word at a time, wait for interrupt for each
;       word.
;--

10$:
        BSBW    MOVFRUSER               ; Get two bytes from user buffer
        DEVICELOCK -
                LOCKADDR=UCB$L_DLCK(R5),- ; Lock device access
                SAVIPL=-(SP),-          ; Save current IPL
                PRESERVE=NO             ; Don't preserve R0
                                        ; Flag interrupt expected
        SETIPL  #31,-                   ; Raise IPL to power
                ENVIRON=UNIPROCESSOR
        MOVW    R1,XA_ODR(R4)           ; Move data to DR11-W
        MOVW    UCB$W_XA_CSRTMP(R5),XA_CSR(R4) ; Set DR11-W CSR
        BBC     #XA$V_LINK,UCB$L_DEVDEPEND(R5),15$ ; Link mode?
        BICW3   #XA$K_FNCT2,UCB$W_XA_CSRTMP(R5),XA_CSR(R4) ;Clear interrupt
                                        ; FNCT bit 2
                                        ; Only if link mode specified
15$:

; Wait for interrupt, powerfail, or device timeout

        WFIKPCH XA_TIME_OUTW,IRP$L_MEDIA(R3)

; Check for errors, decrement transfer count, and loop until complete
```

```
        IOFORK                          ; Fork to lower IPL
        CMPB    #DT$_DR11W,-            ; Branch if this is a DR11-W
                  UCB$B_DEVTYPE(R5)
        BEQL    17$
        BBC     #XA_CSR$V_ERROR,-       ; DRV11-WA - check ERROR bit in CSR.
                  UCB$W_XA_CSR(R5),20$  ; Branch on success.
        BRW     40$                     ; Branch on error.
17$:    BITW    #XA_EIR$M_NEX!-
                XA_EIR$M_MULTI!-
                XA_EIR$M_ACLO!-
                XA_EIR$M_PAR!-
                XA_EIR$M_DLT,UCB$W_XA_EIR(R5) ; Any errors?
        BEQL    20$                     ; No, continue
        BRW     40$                     ; Yes, abort transfer.
20$:    DECW    UCB$L_XA_DPR(R5)                ; All words transferred?
        BNEQ    10$                     ; No, loop until finished.

; Transfer is done, clear iterrupt expected flag and FORK
; All words read or written in WORD MODE.  Finish I/O.

RETURN_STATUS:

        JSB     G^IOC$DIAGBUFILL        ; Fill diagnostic buffer if present
        BSBW    DEL_ATTNAST             ; Deliver outstanding ATTN ASTs
        MOVZWL  #SS$_NORMAL,R0          ; Complete success status
22$:    MULW3   #2,UCB$L_XA_DPR(R5),R1  ; Calculate actual bytes transferred
        SUBW3   R1,UCB$W_BCNT(R5),R1    ; From requested number of bytes
        INSV    R1,#16,#16,R0           ; And place in high word of R0
        MOVL    UCB$W_XA_CSR(R5),R1     ; Return CSR and EIR status
        BISB    #XA_CSR$M_IE,XA_CSR(R4) ; Enable device interrupts
        REQCOM                          ; Finish request in exec

;++
; WORD MODE READ -- Read (input) in word mode
;
; FUNCTIONAL DESCRIPTION:
;
;     Transfer the requested number of words from the DR11-W IDR into
;     user memory one word at a time, wait for interrupt for each word.
;     If the unexpected (unsolicited) interrupt bit is set, transfer the
;     first (last received) word to memory without waiting for an
;     interrupt.
;--

30$:
        DEVICELOCK -
                LOCKADDR=UCB$L_DLCK(R5),- ; Lock device access
                SAVIPL=-(SP),-          ; Save current IPL
                PRESERVE=NO             ; Don't preserve R0

; If an unexpected (unsolicited) interrupt has occurred, assume it
; is for this READ request and return value to user buffer without
; waiting for an interrupt.

        BBCC    #UCB$V_UNEXPT,-
                UCB$W_DEVSTS(R5),32$    ; Branch if no unexpected interrupt

        DEVICEUNLOCK -
                LOCKADDR=UCB$L_DLCK(R5),- ; Unlock device access
                NEWIPL=(SP)+,-          ; Enable interrupts
                PRESERVE=NO
        BRB     37$                     ; continue

32$:
        SETIPL  #IPL$_POWER,-
                ENVIRON=UNIPROCESSOR
```

```
35$:
; Wait for interrupt, powerfail, or device timeout
        WFIKPCH XA_TIME_OUTW,IRP$L_MEDIA(R3)
; Check for errors, decrement transfer count and loop until done
        IOFORK                          ; Fork to lower IPL
37$:
        CMPB    #DT$_DR11W,-            ; Branch if this is a DR11-W
                  UCB$B_DEVTYPE(R5)
        BEQL    1037$
        BBC     #XA_CSR$V_ERROR,-       ; DRV11-WA - check ERROR bit in CSR.
                  UCB$W_XA_CSR(R5),1038$ ; Branch on success.
        BRW     40$                     ; Branch on error.
1037$:  BITW    #XA_EIR$M_NEX!-
                XA_EIR$M_MULTI!-
                XA_EIR$M_ACLO!-
                XA_EIR$M_PAR!-
                XA_EIR$M_DLT,UCB$W_XA_EIR(R5) ; Any errors?
        BNEQ    40$                     ; Yes, abort transfer.
1038$:  BSBW    MOVTOUSER               ; Store two bytes into user buffer
; Send interrupt back to sender.  Acknowledge we got last word.
        DEVICELOCK -
                LOCKADDR=UCB$L_DLCK(R5),- ; Lock device access
                SAVIPL=-(SP),-          ; Save current IPL
                PRESERVE=NO             ; Don't preserve R0
        MOVW    UCB$W_XA_CSRTMP(R5),XA_CSR(R4)
        BBC     #XA$V_LINK,UCB$L_DEVDEPEND(R5),38$ ; Link mode?
        BICW3   #XA$K_FNCT2,UCB$W_XA_CSRTMP(R5),XA_CSR(R4) ;Yes, clear
                                                           ; FNCT 2
38$:
        DECW    UCB$L_XA_DPR(R5)                ; Decrement transfer count
        BNEQ    35$                     ; Loop until all words transferred
        DEVICEUNLOCK -
                LOCKADDR=UCB$L_DLCK(R5),- ; Unlock device access
                NEWIPL=(SP)+,-          ; Enable interrupts
                PRESERVE=NO
        BRW     RETURN_STATUS           ; Finish request in common code
; Error detected in word mode transfer
40$:
        BSBW    DEL_ATTNAST             ; Deliver ATTN ASTs
        BSBW    XA_DEV_RESET            ; Error, reset DR11-W
        JSB     G^IOC$DIAGBUFILL        ; Fill diagnostic buffer if present
        JSB     G^ERL$DEVICERR          ; Log device error
        MOVZWL  UCB$W_XA_ERROR(R5),R0   ; Set controller/drive status in R0
        BRW     22$
```

```
        .DSABL  LSB
;
; MOVFRUSER - Routine to fetch two bytes from user buffer.
;
; INPUTS:
;
;       R5 = UCB address
;
; OUTPUTS:
;
;       R1 = Two bytes of data from user's buffer
;       Buffer descriptor in UCB is updated.
;
        .ENABL  LSB
MOVFRUSER:
        MOVAL   -(SP),R1                ; Address of temporary stack loc
        MOVZBL  #2,R2                   ; Fetch two bytes
        JSB     G^IOC$MOVFRUSER         ; Call exec routine to do the deed
        MOVL    (SP)+,R1                ; Retrieve the bytes
        BRB     20$                     ; Update UCB buffer pointers
;
; MOVTOUSER - Routine to store two bytes into user's buffer.
;
; INPUTS:
;
;       R5 = UCB address
;       UCB$W_XA_IDR(R5) = Location where two bytes are saved
;
; OUTPUTS:
;
;       Two bytes are stored in user buffer and buffer descriptor in
;       UCB is updated.
;
MOVTOUSER:
        MOVAB   UCB$W_XA_IDR(R5),R1     ; Address of internal buffer
        MOVZBL  #2,R2
        JSB     G^IOC$MOVTOUSER         ; Call exec
20$:                                    ; Update buffer pointers in UCB
        ADDW    #2,UCB$W_BOFF(R5)       ; Add two to buffer descriptor
        BICW    #^C<^X01FF>,UCB$W_BOFF(R5) ; Modulo the page size
        BNEQ    30$                     ; If NEQ, no page boundary crossed
        ADDL    #4,UCB$L_SVAPTE(R5)     ; Point to next page
30$:
        RSB
;
        .DSABL  LSB

        .PAGE
        .SBTTL  DR11-W DEVICE TIMEOUT
;++
; DR11-W device TIMEOUT
; If a DMA transfer was in progress, release UBA resources.
; For DMA or WORD mode, deliver attention ASTs, log a device timeout error,
; and do a hard reset on the controller.
;
; Clear DR11-W CSR
; Return error status
;
; Power failure will appear as a device timeout
;--
        .ENABL  LSB
XA_TIME_OUT:                            ; Timeout for DMA transfer
```

```
        BSBW    XA_DEV_HWRESET          ; reset h/w &
                                        ; blow away any outstanding DMA

        IOFORK                          ; Fork to complete request
        PURDPR                          ; Purge buffered data path in UBA
        RELMPR                          ; Release UBA map registers
        RELDPR                          ; Release UBA data path
        BRB     10$                     ; continue

XA_TIME_OUTW:                           ; Timeout for WORD mode transfer

        IOFORK                          ; Fork to complete operations
10$:    MOVL    UCB$L_CRB(R5),R4        ; Fetch address of CSR
        MOVL    @CRB$L_INTD+VEC$L_IDB(R4),R4
        BSBW    XA_REGISTER             ; Read DR11-W registers
        JSB     G^IOC$DIAGBUFILL        ; Fill diagnostic buffer
        JSB     G^ERL$DEVICTMO          ; Log device time out
        BSBW    DEL_ATTNAST             ; And deliver the ASTs
        BSBW    XA_DEV_RESET            ; Reset controller
        MOVZWL  #SS$_TIMEOUT,R0         ; Assume error status
        BBC     #UCB$V_CANCEL,-
                UCB$W_STS(R5),20$       ; Branch if not cancel
        MOVZWL  #SS$_CANCEL,R0          ; Set status
20$:    CLRL    R1
        BICW    #UCB$M_ATTNAST!UCB$M_UNEXPT,UCB$W_DEVSTS(R5)
                                        ; Clear unwanted flags.
        BICW    #<UCB$M_TIM!UCB$M_INT!UCB$M_TIMOUT!UCB$M_CANCEL!UCB$M_POWER>,-
                UCB$W_STS(R5)           ; Clear unit status flags
        REQCOM                          ; Complete I/O in exec
        .DSABL  LSB
        .PAGE

        .SBTTL  XA_INTERRUPT, Interrupt service routine for DR11-W
;++
; XA_INTERRUPT, Handles interrupts generated by DR11-W
;
; Functional description:
;
;       This routine is entered whenever an interrupt is generated
;       by the DR11-W.  It checks that an interrupt was expected.
;       If not, it sets the unexpected (unsolicited) interrupt flag.
;       All device registers are read and stored into the UCB.
;       If an interrupt was expected, it calls the driver back at its Wait
;       For Interrupt point.
;       Deliver attention ASTs if unexpected interrupt.
;
; Inputs:
;
;       00(SP) = Pointer to address of the device IDB
;       04(SP) = saved R0
;       08(SP) = saved R1
;       12(SP) = saved R2
;       16(SP) = saved R3
;       20(SP) = saved R4
;       24(SP) = saved R5
;       28(SP) = saved PSL
;       32(SP) = saved PC
;
; Outputs:
;
;       The driver is called at its Wait For Interrupt point if an
;       interrupt was expected.
;       The current value of the DR11-W CSRs are stored in the UCB.
;
```

```
;--
XA_INTERRUPT:                           ; Interrupt service for DR11-W
        MOVL    @(SP)+,R4               ; Address of IDB and pop SP
        MOVQ    (R4),R4                 ; CSR and UCB address from IDB

        DEVICELOCK -
                LOCKADDR=UCB$L_DLCK(R5),- ; Lock device access
                CONDITION=NOSETIPL,-    ; already at DIPL
                PRESERVE=NO             ; Don't preserve R0

; Read the DR11-W device registers (WCR, BAR, CSR, EIR, IDR) and store
; into UCB.

        BSBW    XA_REGISTER             ; Read device registers

; Check to see if device transfer request active or not
; If so, call driver back at Wait for Interrupt point and
; Clear unexpected interrupt flag.

20$:    BBCC    #UCB$V_INT,UCB$W_STS(R5),25$
                                        ; If clear, no interrupt expected

; Interrupt expected, clear unexpected interrupt flag and call driver
; back.

        BICW    #UCB$M_UNEXPT,UCB$W_DEVSTS(R5)
                                        ; Clear unexpected interrupt flag
        MOVL    UCB$L_FR3(R5),R3        ; Restore driver's R3
        JSB     @UCB$L_FPC(R5)          ; Call driver back
        BRB     30$

; Deliver attention ASTs if no interrupt expected and set unexpected
; interrupt flag.

25$:    BBSC    #UCB$V_IGNORE_UNEXPT,-  ; Ignore spurious interrupt -
                  UCB$W_DEVSTS(R5),24$  ;   (DRV11-WA only.)
        BISW    #UCB$M_UNEXPT,UCB$W_DEVSTS(R5) ;Set unexpected interrupt flag
        BSBW    DEL_ATTNAST             ; Deliver attention ASTs
        BISB    #XA_CSR$M_IE,XA_CSR(R4) ; Enable device interrupts
        BRB     30$

; Restore registers and return from interrupt

24$:    NOP             ; allow a breakpoint here (spurious interrupt)

30$:    DEVICEUNLOCK -
                LOCKADDR=UCB$L_DLCK(R5),- ; Unlock device access
                PRESERVE=NO
        POPR    #^M<R0,R1,R2,R3,R4,R5>  ; Restore registers
        REI                             ; Return from interrupt
        .PAGE
```

```
        .SBTTL  XA_REGISTER - Handle DR11-W CSR transfers
;++
; XA_REGISTER - Routine to handle DR11-W register transfers
;
; INPUTS:
;
;       R4 - DR11-W CSR address
;       R5 - UCB address of unit
;
; OUTPUTS:
;
;       CSR, EIR, WCR, BAR, BAE, IDR, and status are read and stored into UCB.
;       The DR11-W is placed in its initial state with interrupts enabled.
;       R0 - .true. if no hard error
;            .false. if hard error (cannot clear ATTN)
;
; If the CSR ERROR bit is set and the associated condition can be cleared,
; then the error is transient and recoverable.  The status returned is
; SS$_DRVERR.  If the CSR ERROR bit is set and cannot be cleared by
; clearing the CSR, then this is a hard error and cannot be recovered.
; The returned status is SS$_CTRLERR.
;
;       R0,R1 - destroyed, all other registers preserved.
;--

XA_REGISTER:

        MOVZWL  #SS$_NORMAL,R0          ; Assume success
        MOVZWL  XA_CSR(R4),R1           ; Read CSR
        MOVW    R1,UCB$W_XA_CSR(R5)     ; Save CSR in UCB
        BBC     #XA_CSR$V_ERROR,R1,55$  ; Branch if no error
        MOVZWL  #SS$_DRVERR,R0          ; Assume "drive" error
55$:    BICW    #^C<XA_CSR$M_FNCT>,R1   ; Clear all uninteresting bits
                                        ; for later
        CMPB    #DT$_XA_DRV11WA,-       ; If this is a DRV11-WA,
                  UCB$B_DEVTYPE(R5)     ;
        BEQL    57$                     ;   then branch.

;
; The guide to programming DRx11's stresses over and over again that clearing
; the ATTN bit (by writing a 0 to it) can lead to a lost interrupt (i.e., a
; pending interrupt request can be flushed). We will handle this case by
; always setting the ATTN bit in the CSR/EIR- which is benign.
;
        BISB    #<XA_CSR$M_ERROR!XA_CSR$M_ATTN/256>,XA_CSR+1(R4) ;Set EIR flag
        MOVW    XA_EIR(R4),UCB$W_XA_EIR(R5) ; Save EIR in UCB
        BRB     59$
57$:    BISW    #XA_CSR$M_IE,R1         ; On the DRV11-WA, if the IE bit
                                        ; makes a 0->1 transition while
                                        ; READY=1, a spurious interrupt
                                        ; in generated.   Therefore, we
                                        ; leave IE high at all times.
;
; The guide to programming DRx11's stresses over and over again that
; clearing the ATTN bit (by writing a 0 to it) can lead to a lost
; interrupt (i.e., a pending interrupt request can be flushed). We will
; handle this case by always setting the ATTN bit in the CSR/EIR- which
; is benign.
;
59$:    BISW3   #XA_CSR$M_ATTN,R1,XA_CSR(R4) ; Clear EIR flag and errors
        MOVW    XA_CSR(R4),R1           ; Read CSR back
60$:    MOVW    XA_IDR(R4),UCB$W_XA_IDR(R5) ; Save IDR in UCB
        MOVW    XA_BAR(R4),UCB$W_XA_BAR(R5)
        CMPB    #DT$_DR11W,-            ; If this is a DR11-W,
```

```
                UCB$B_DEVTYPE(R5)       ;
        BEQL    70$                     ;   then branch.
        MOVW    XA_BAE(R4),UCB$W_XA_BAE(R5) ; Save BAE in UCB
70$:    MOVW    XA_WCR(R4),UCB$W_XA_WCR(R5)
        MOVW    R0,UCB$W_XA_ERROR(R5)   ; Save status in UCB
        RSB

        .SBTTL  XA_CANCEL, Cancel I/O routine
;++
; XA_CANCEL, Cancels an I/O operation in progress
;
; Functional description:
;
;       Flushes Attention AST queue for the user.
;       If transfer in progress, do a device reset to DR11-W and finish the
;       request.
;       Clear interrupt expected flag.
;
; Inputs:
;
;       R2 = negated value of channel index
;       R3 = address of current IRP
;       R4 = address of the PCB requesting the cancel
;       R5 = address of the device's UCB
;
; Outputs:
;
;--
XA_CANCEL:                                      ; Cancel I/O

        DEVICELOCK -
                LOCKADDR=UCB$L_DLCK(R5),- ; Lock device access
                SAVIPL=-(SP),-          ; Save current IPL
                PRESERVE=NO             ; Don't preserve R0
        BBCC    #UCB$V_ATTNAST,-
                UCB$W_DEVSTS(R5),20$    ; attention AST enabled?

; Finish all attention ASTs for this process.

        PUSHR   #^M<R2,R6,R7>
        MOVL    R2,R6                   ; Set up channel number
        MOVAB   UCB$L_XA_ATTN(R5),R7    ; Address of listhead
        JSB     G^COM$FLUSHATTNS        ; Flush attention ASTs for process
        POPR    #^M<R2,R6,R7>

; Check to see if a data transfer request is in progress
; for this process on this channel
```

```
20$:    BBC     #UCB$V_INT,-            ; Branch if I/O not in progress
                UCB$W_STS(R5),30$
        JSB     G^IOC$CANCELIO          ; Check if transfer going
        BBC     #UCB$V_CANCEL,-
                UCB$W_STS(R5),30$       ; Branch if not for this guy
;
; Force timeout
;
        CLRL    UCB$L_DUETIM(R5)        ; Clear timer
        BISW    #UCB$M_TIM,UCB$W_STS(R5) ; set timed
        BICW    #UCB$M_TIMOUT,-
                UCB$W_STS(R5)           ; Clear timed out
30$:
        DEVICEUNLOCK -
                LOCKADDR=UCB$L_DLCK(R5),- ; Unlock device access
                NEWIPL=(SP)+,-          ; Enable interrupts
                PRESERVE=NO
        RSB                             ; Return

        .PAGE
        .SBTTL  DEL_ATTNAST, Deliver attention ASTs
;++
; DEL_ATTNAST, Deliver all outstanding attention ASTs
;
; Functional description:
;
;    This routine is used by the DR11-W driver to deliver all of the
;    outstanding attention ASTs.  It is copied from COM$DELATTNAST in
;    the exec.  In addition, it places the saved value of the DR11-W CSR
;    and Input Data Buffer Register in the AST parameter.
;
; Inputs:
;
;       R5 = UCB of DR11-W unit
;
; Outputs:
;
;       R0,R1,R2 Destroyed
;       R3,R4,R5 Preserved
;--
DEL_ATTNAST:
        DEVICELOCK -
                LOCKADDR=UCB$L_DLCK(R5),- ; Lock device access
                SAVIPL=-(SP),-          ; Save current IPL
                PRESERVE=NO             ; Don't preserve R0
        BBCC    #UCB$V_ATTNAST,UCB$W_DEVSTS(R5),30$
                                        ; Any attention ASTs expected?
        PUSHR   #^M<R3,R4,R5>           ; Save R3,R4,R5
10$:    MOVL    8(SP),R1                ; Get address of UCB
        MOVAB   UCB$L_XA_ATTN(R1),R2    ; Address of attention AST listhead
        MOVL    (R2),R5                 ; Address of next entry on list
        BEQL    20$                     ; No next entry, end of loop
        BICW    #UCB$M_UNEXPT,UCB$W_DEVSTS(R1) ;Clear unexpected interrupt
                                                ; flag
        MOVL    (R5),(R2)               ; Close list
        MOVW    UCB$W_XA_IDR(R1),ACB$L_KAST+6(R5)
                                        ; Store IDR in AST parameter
        MOVW    UCB$W_XA_CSR(R1),ACB$L_KAST+4(R5)
                                        ; Store CSR in AST parameter
        PUSHAB  B^10$                   ; Set return address for FORK
        FORK                            ; FORK for this AST

; AST fork procedure
```

```
        MOVQ    ACB$L_KAST(R5),ACB$L_AST(R5)
                                        ; Rearrange entries
        MOVB    ACB$L_KAST+8(R5),ACB$B_RMOD(R5)
        MOVL    ACB$L_KAST+12(R5),ACB$L_PID(R5)
        CLRL    ACB$L_KAST(R5)
        MOVZBL  #PRI$_IOCOM,R2          ; Set up priority increment
        JMP     G^SCH$QAST              ; Queue the AST

20$:    POPR    #^M<R3,R4,R5>           ; Restore registers
30$:    DEVICEUNLOCK -
                LOCKADDR=UCB$L_DLCK(R5),- ; Unlock device access
                NEWIPL=(SP)+,-          ; Enable interrupts
                PRESERVE=NO,-           ;
                CONDITION=RESTORE       ;
        RSB                             ; Return

        .PAGE
        .SBTTL  XA_REGDUMP - DR11-W register dump routine
;++
; XA_REGDUMP - DR11-W Register dump routine.
;
; This routine is called to save the controller registers in a specified
; buffer.  It is called from the device error logging routine and from the
; diagnostic buffer fill routine.
;
; Inputs:
;
;       R0 - Address of register save buffer
;       R4 - Address of Control and Status Register
;       R5 - Address of UCB
;
; Outputs:
;
;       The controller registers are saved in the specified buffer.
;
;               CSRTMP - The last command written to the DR11-W CSR by
;                        by the driver.
;               BARTMP - The last value written into the DR11-W BAR by
;                        the driver during a block mode transfer.
;               CSR - The CSR image at the last interrupt
;               EIR - The EIR image at the last interrupt
;               IDR - The IDR image at the last interrupt
;               BAR - The BAR image at the last interrupt
;               WCR - Word count register
;               ERROR - The system status at request completion
;               PDRN - UBA Datapath Register number
;               DPR - The contents of the UBA Data Path register
;               FMPR - The contents of the last UBA Map register
;               PMRP - The contents of the previous UBA Map register
;               DPRF - Flag for purge datapath error
;                       0 = no purger datapath error
;                       1 = parity error when datapath was purged
;               BAETMP - The last value written to the BAE by the
;                        driver during a block mode transfer (DRV11-WA only)
;               BAE - The BAE image at the last interrupt (DRV11-WA only)
;
;       Note that the values stored are from the last completed transfer
;       operation. If a zero transfer count is specified, then the
;       values are from the last operation with a non-zero transfer count.
;--

XA_REGDUMP:
```

```
        MOVZBL  #15,(R0)+               ; 15 registers are stored.
        MOVAB   UCB$W_XA_CSRTMP(R5),R1  ;Get address of saved register images
        MOVZBL  #8,R2                   ; Return 8 registers here
10$:    MOVZWL  (R1)+,(R0)+
        SOBGTR  R2,10$                  ; Move them all
        MOVZBL  UCB$W_XA_DPRN(R5),(R0)+ ; Save Datapath Register number
        MOVZBL  #3,R2                   ; And 3 more here
20$:    MOVL    (R1)+,(R0)+             ; Move UBA register contents
        SOBGTR  R2,20$
        MOVZBL  UCB$W_XA_DPRN+1(R5),(R0)+ ; Save Datapath Parity Error Flag
        MOVZWL  UCB$W_XA_BAETMP(R5),(R0)+ ; Save BAE stored prior to xfer
        MOVZWL  UCB$W_XA_BAE(R5),(R0)+  ; Save BAE store following xfer
        RSB

        .PAGE
        .SBTTL  XA_DEV_RESET - Device reset DR11-W
;++
; XA_DEV_RESET - DR11-W Device reset routine
;
; This routine raises IPL to device IPL, performs a device reset to
; the required controller, and reenables device interrupts.
;
; Must be called at or below device IPL to prevent a confict in
; acquiring the device spinlock.
;
; Inputs:
;
;       R4 - Address of Control and Status Register
;       R5 - Address of UCB
;
; Outputs:
;
;       Controller is reset, controller interrupts are enabled
;
;--

XA_DEV_RESET:

        PUSHR   #^M<R0,R1,R2>           ; Save some registers
        DEVICELOCK -
                LOCKADDR=UCB$L_DLCK(R5),- ; Lock device access
                SAVIPL=-(SP),-          ; Save current IPL
                PRESERVE=NO             ; Don't preserve R0

        BSBB    XA_DEV_HWRESET

        DEVICEUNLOCK -
                LOCKADDR=UCB$L_DLCK(R5),- ; Unlock device access
                NEWIPL=(SP)+,-          ; Enable interrupts
                PRESERVE=NO
        POPR    #^M<R0,R1,R2>           ; Restore registers

        RSB

XA_DEV_HWRESET:

        CMPB    #DT$_DR11W,-            ; If this is a DR11-W,
                  UCB$B_DEVTYPE(R5)     ;
        BEQL    20$                     ;   then branch.
        MOVW    #XA_CSR$M_IE,XA_CSR(R4) ; Clear all writable bits but IE.
        BITB    #XA_CSR$M_RDY,XA_CSR(R4); If not ready then no transfer
                                        ; in progress,
        BNEQ    40$                     ; so no need to reset device
```

```
        MNEGW   #1,XA_WCR(R4)           ;Tell it only 1 byte left to transfer
        BISB    #<XA_CSR$M_CYCLE!XA_CSR$M_ATTN>/256,-
                  XA_CSR+1(R4)          ; and complete the transfer.
        CLRW    XA_CSR(R4)              ; Clear any errors
        BRB     30$
20$:    MOVB    #<XA_CSR$M_MAINT/256>,XA_CSR+1(R4)
        CLRB    XA_CSR+1(R4)

; ***  Must delay here depending on reset interval

30$:    TIMEDWAIT TIME=#XA_RESET_DELAY  ; No. of 10 micro-sec intervals
                                        ; to wait
         MOVB    #XA_CSR$M_IE,XA_CSR(R4) ; Reenable device interrupts
40$:    RSB

XA_END:                                         ; End of driver label
        .END
```

E Multiprocessing Requirements on Kernel-Mode Code

Several features introduced in recent VMS releases have some impact on the execution of non-Digital-supplied kernel-mode code, most notably device drivers written prior to VMS Version 5.0. This chapter describes those changes Digital requires or recommends in an existing non-Digital-supplied device driver. It also provides a brief explanation of key VMS concepts that are integral to an understanding of the operation of privileged code in recent VMS versions.

E.1 Uniprocessor and Multiprocessor Device Drivers

One of the most significant components of VMS introduced at Version 5.0 is its support of a symmetric multiprocessing environment for certain VAX systems, including the VAX 8300/8350, VAX 8800/8820 /8830, VAX 6000, and VAX 9000 series. The multiprocessing environment provided by earlier versions of VMS was asymmetric in nature. Because only the primary processor could execute kernel-mode code, kernel-mode code, including device drivers, effectively ran in a uniprocessing environment and did not need to undertake any special actions due to the multiprocessing nature of the system.

However, in the new symmetric multiprocessing environment supported by VMS beginning at Version 5.0, all processors in the system can execute kernel-mode code. Consequently, privileged code must take steps to ensure that its execution and use of memory are synchronized with kernel-mode code that may be executing concurrently on another processor. Such code must maintain two dimensions of synchronization: raising to the appropriate IPL for a certain transaction, while securing the proper spin lock for the object of that transaction.

For privileged code executing within a VMS uniprocessing environment, VMS now transparently forgoes the second of these requirements. That is, on a VAX uniprocessor, or in a VMS multiprocessor system wherein multiprocessing is *not* enabled, privileged code may securely execute by adhering to the IPL synchronization method alone.

To support both uniprocessor and multiprocessor environments in the most efficient and secure way possible, VMS now incorporates special logic in the System Generation Utility (SYSGEN), the device driver loading mechanism, and several synchronization macros. This code enables VMS to discern the environment in which it is executing and, most importantly, to take steps to prohibit a privileged code thread from executing without proper synchronization in a multiprocessing environment.

As discussed in Section E.3, non-Digital-supplied device drivers written prior to VMS Version 5.0 *must* be altered to execute correctly in a VMS symmetric multiprocessing environment. The modifications discussed in Section E.3 are not required for a device driver that will be loaded and executed *only* on a VMS uniprocessor system. However, the same macros,

routines, and field names used in a multiprocessing environment are accepted by VMS in a uniprocessing environment. Furthermore, the spin lock synchronization macros and routines are specially designed to execute a streamlined code that obtains IPL synchronization alone in such an environment. Digital recommends that any driver that may execute in a multiprocessing environment be updated accordingly.

The remainder of this section identifies the activities of the MULTIPROCESSING system parameter, VMS driver loading mechanisms, and the VMS synchronization macros in creating a multiprocessing or uniprocessing environment and enforcing the appropriate synchronization.

E.1.1 MULTIPROCESSING System Parameter

Every VMS system is initially booted as a single processor, regardless of its hardware configuration. The setting of the MULTIPROCESSING system parameter for the first processor in the system to boot (called the **primary processor** in a multiprocessing environment) determines which synchronization image the secondary bootstrap program (SYSBOOT) loads into memory as part of the operating system. Table E–1 describes the contents of the three possible synchronization images.

Table E–1 VMS Synchronization Images

Image	Results
Uniprocessing	Synchronization is accomplished by elevating IPL. Spin lock acquisition routines only achieve IPL synchronization.
Full-checking	Synchronization is accomplished by both elevating IPL and obtaining an appropriate spin lock. Spin lock acquisition routines perform both of these tasks. Spin lock acquisition routines also perform spin lock rank checking and verify the spin lock synchronization IPL, issuing appropriate bugchecks if they discover violations of synchronization rules. Spin lock acquisition routines maintain various debugging aids and performance analysis aids (such as the longwords in the spin lock data structure containing the PCs of the most recent acquisitions and releases of the spin lock and the set of counters in the per-CPU database structure (CPU)). (See Section E.3.7 for additional description of full-checking synchronization.)
Streamlined	Synchronization is accomplished by both elevating IPL and obtaining an appropriate spin lock. Spin lock acquisition routines do *not* perform checking and do *not* record the PCs of the spin lock acquisitions and releases.

Table E–2 lists the possible settings of the MULTIPROCESSING system parameter.

Table E–2 Settings of MULTIPROCESSING System Parameter

Value	Result
0	Loads uniprocessing synchronization image for any hardware configuration
1	Loads full-checking synchronization image and sets multiprocessing-enabled bit (SMP$V_ENABLED in SMP$GL_FLAGS) if the hardware configuration is capable of multiprocessing and two or more processors are available; otherwise, loads uniprocessing synchronization image.
2	Loads full-checking synchronization image and sets multiprocessing-enabled bit regardless of the hardware configuration.
3	Loads streamlined synchronization image and sets multiprocessing-enabled bit if the hardware configuration is capable of multiprocessing and two or more processors are available; otherwise, loads uniprocessing synchronization image. This is the default value.

E.1.2 Device Driver Loading

In a VMS multiprocessing environment, the presence of a device driver that does not adhere to multiprocessing synchronization conventions introduced at VMS Version 5.0 can be fatal to proper system functions. Now, VMS takes steps to either prohibit the enabling of multiprocessing in a VAX system that has such a driver present or prevent the loading of such a driver if multiprocessing has already been enabled.

To accomplish this, the VMS driver-loading routine assumes that any driver that can run in a VMS multiprocessing environment uses the spin lock synchronization macros and loads the appropriate I/O database fields. (See Section E.3 for information on how to produce a driver that can execute in a VMS multiprocessing environment.) Use of the spin lock synchronization macros causes VMS to set the SMP-modified bit in the DPT (DPT$V_SMPMOD in DPT$L_FLAGS).

If multiprocessing has *not* been enabled on the system, the driver loading mechanism checks the SMP-modified bit in the DPT and takes either of the following actions:

- If the SMP-modified bit is set, the driver loading mechanism loads the driver and calls its controller and unit initialization routines, as discussed in Chapter 12.
- If the SMP-modified bit is *not* set, the driver loading mechanism sets the unmodified-driver bit (SMP$V_UNMOD_DRIVER) in SMP$GL_FLAGS, thus prohibiting the subsequent enabling of multiprocessing on the system. It then loads the driver and calls its controller and unit initialization routines, as described in Chapter 12. If such a driver has been successfully loaded into a VMS system, you cannot subsequently enable multiprocessing.

If multiprocessing is currently enabled on the system, the driver loading mechanism checks the SMP-modified bit in the DPT and takes either of the following actions:

- If the SMP-modified bit is set, the driver loading mechanism loads the driver and calls its controller and unit initialization routines, as discussed in Chapter 12.
- If the SMP-modified bit is *not* set, the driver loading mechanism does not load the driver, returning the error status SS$_NONSMPDRV to its caller.

E.1.3 VMS Synchronization Macros

To support the spin lock synchronization required in VMS multiprocessor systems, VMS Version 5.0 added the DEVICELOCK/DEVICEUNLOCK, FORKLOCK/FORKUNLOCK, and LOCK/UNLOCK macros to the existing SETIPL and DSBINT/ENBINT macros. As discussed in Section E.3, the SETIPL and DSBINT/ENBINT macros must not be used to synchronize systemwide activities in a VMS multiprocessing environment. However, the DEVICELOCK/DEVICEUNLOCK, FORKLOCK/FORKUNLOCK, and LOCK/UNLOCK macros are designed to operate appropriately in either a multiprocessing or uniprocessing environment. According to the value of the multiprocessing-enabled bit (SMP$V_ENABLED) in SMP$GL_FLAGS, the run-time code produced by these macros behaves as follows:

- If multiprocessing has *not* been enabled in the system, these macros only raise or lower IPL to the IPL required to synchronize access to the specified system resource.
- If multiprocessing has been enabled in the system, these macros call the appropriate spin lock synchronization routine, which acquires or releases the spin lock corresponding to the system resource, raising or lowering IPL as required.

The setting of the MULTIPROCESSING system parameter controls the disposition of the multiprocessing-enabled bit, as discussed in Section E.1.1. The macro chapter in the *VMS Device Support Reference Manual* describes the VMS synchronization macros in full.

E.2 Changes Required to Drivers Written Before VMS Version 5.0

Most changes introduced at VMS Version 5.0 are transparent to non-Digital-supplied drivers written prior to that release and can be accommodated in the driver image by simply reassembling and relinking the driver. However, there are several required—and some recommended—changes that the writers and maintainers of these drivers should make before attempting these tasks. This section describes these modifications.

In addition, if a non-Digital-supplied driver is to be loaded and run in a VMS symmetric multiprocessing system, it is critical that it be adapted according to the guidelines discussed in Section E.3. Failure to adapt such drivers to use multiprocessing synchronization mechanisms may result in

either a failure to load the driver or the inability to enable multiprocessing on the system.

E.2.1 Address of the Driver's Interrupt Service Routine in the DPT

In order to provide an optional method of servicing MicroVAX Q22 bus device interrupts at the IPLs at which they are requested, VMS now defines several new symbolic offsets in the interrupt dispatch vector (VEC) portion of the channel request block (CRB).

One of these symbolic offsets is significant to all device drivers. Prior to VMS Version 5.0, device drivers initialized the location in the vector containing the address of the driver's interrupt service routine by referring explicitly to its location (CRB$L_INTD+4). Digital now recommends that all device drivers refer to this location using the symbolic offset CRB$L_INTD+VEC$L_ISR, as follows:

Old: DPT_STORE CRB,CRBL_INTD+4,D,LPINT_SERV_RTN

New: DPT_STORE CRB,CRB$L_INTD+VEC$L_ISR,D,LP$INT_SERV_RTN

To use the new symbols, you must include the $CRBDEF and $VECDEF structure definition macros in the driver. All structure definition macros can be found in SYS$LIBRARY:LIB.MLB.

E.2.2 Checking, Debiting, and Crediting a Process's Byte Count Quota

Now the routines EXE$BUFFRQUOTA and EXE$BUFQUOPRC are replaced with a set of eight new routines that manipulate a job's byte count quota and byte limit, optionally allocating a nonpaged pool buffer of the requested size. To ensure proper synchronization, programs should use these routines and avoid any direct manipulation of the nonpaged pool quota fields JIB$L_BYTCNT and JIB$L_BYTLM).

Among the new routines are the following:

Routine	Function
EXE$CREDIT_BYTCNT	Returns credit to a job's byte count quota
EXE$CREDIT_BYTCNT_BYTLM	Returns credit to a job's byte count quota and byte count limit
EXE$DEBIT_BYTCNT	Determines whether a job's buffered byte count quota usage permits the process to be granted additional buffered I/O and, if so, adjusts the job's byte count quota
EXE$DEBIT_BYTCNT_NW	Same function as EXE$DEBIT_BYTCNT, but never places a process in a resource wait state pending the return of sufficient quota

Routine	Function
EXE$DEBIT_BYTCNT_BYTLM	Determines whether a job's buffered byte count quota usage permits the process to be granted additional buffered I/O and, if so, adjusts the job's byte count quota and byte count limit
EXE$DEBIT_BYTCNT_BYTLM_NW	Same function as EXE$DEBIT_BYTCNT_BYTLM, but never places a process in a resource wait state pending sufficient quota
EXE$DEBIT_BYTCNT_ALO	Same function as EXE$DEBIT_BYTCNT, but, if quota checks succeed, allocates the requested amount of pool
EXE$DEBIT_BYTCNT_BYTLM_ALO	Same function as EXE$DEBIT_BYTCNT_BYTLM, but, if quota checks succeed, allocates the requested amount of pool

Many drivers written prior to VMS Version 5.0 contain code sequences similar to the following:

```
        JSB     G^EXE$BUFFRQUOTA        ;Would buffer allocation
                                        ;exceed byte count quota?
        BLBC    R0,ERROR                ;Branch if yes
        JSB     G^EXE$ALLOCBUF          ;If not, allocate buffer
        BLBC    R0,ERROR                ;Branch if error
        MOVL    PCB$L_JIB(R4),R5        ;Obtain job information
                                        ;block
        SUBL2   R1,JIB$L_BYTCNT(R5)     ;Decrement job's byte count
                                        ;quota
.
.
.
```

The new routines allow you to simplify such code sequences. For instance, a single routine, EXE$DEBIT_BYTCNT_ALO, checks and debits quotas and allocates pool. When there is not enough quota available to service the request, the routine restores the deducted amount and returns the error SS$_EXQUOTA IN R0.

The preceding code example can be rewritten as follows:

```
        JSB     G^EXE$DEBIT_BYTCNT_ALO  ;Check for quota violation,
                                        ;allocate buffer, decrement
                                        ;JIB byte count quota
        BLBC    R0,ERROR                ;Branch if error
.
.
.
```

E.2.3 Referring to the Current PCB

The symbol SCH$GL_CURPCB is obsolete and should be replaced as follows:

- If the process's P1 space is available, use the P1 space location CTL$GL_PCB.

- If the process's P1 space is *not* available, use the FIND_CPU_DATA macro, as follows:

```
FIND_CPU_DATA  R0
MOVL    CPU$L_CURPCB(R0),R1
```

The FIND_CPU_DATA macro obtains the virtual address of the per-CPU database for the processor on which it executes. Code that issues the FIND_CPU_DATA macro must adhere to the following rules:

— It must be executing in kernel mode above IPL 2 when it invokes the FIND_CPU_DATA macro.

— It must take care to prevent rescheduling after issuing the macro as long as the information returned by FIND_CPU_DATA is in use. It typically does this by remaining at an IPL greater than 2.

E.2.4 Allocating System Page-Table Entries

The system routine LDR$ALLOC_PT replaces IOC$ALLOSPT (since Version 5.0) as the appropriate method for non-Digital-supplied VAXBI device drivers to map a portion of a device's node space to system virtual address space. See the routines chapter in the *VMS Device Support Reference Manual* for a full description of LDR$ALLOC_PT.

E.2.5 Referring to a System Process Mailbox

An existing driver that refers to either the job controller's mailbox or OPCOM's mailbox must be altered to use the new symbolic names that point to these mailboxes. Usually, it is the driver's interrupt service routine or timeout handling routine that loads the address of the mailbox UCB into R3 and calls the system routine EXE$SNDEVMSG, as follows.

```
MOVL    G^SYS$AR_JOBCTLMB,R3    ;Set address of job controller
                                ;mailbox
JSB     G^EXE$SNDEVMSG          ;Sent message to job con-
                                ;troller
```

The new symbolic names actually refer to global pointers to the mailbox UCB structures. They include the following:

Since VMS V5.0	Prior to V5.0	Name
SYS$AR_JOBCTLMB	SYS$GL_JOBCTLMB	Job controller's mailbox
SYS$AR_OPRMBX	SYS$GL_OPRMBX	OPCOM's mailbox

E.2.6 Reassembling and Relinking the Driver

Because of changes in the definitions of data structures, the behavior of system macros, the location of global symbols, and the contents of system images, it is necessary to reassemble and relink non-Digital-supplied drivers regardless of whether their contents have been modified.

To do so, reassemble your driver against SYS$LIBRARY:LIB.MLB. For example:

```
$ MACRO MYDRIVER.MAR+SYS$LIBRARY:LIB.MLB/LIBRARY
```

Relink your driver against the VMS global symbol table. If the driver consists of several source files, you must specify the file that contains the driver prologue table as the first file in the list. The linker options file must contain the statement BASE=0. For example:

```
$ CREATE MYDRIVER.OPT
BASE=0
Ctrl/Z
$ LINK/NOTRACE MYDRIVER1[,MYDRIVER2,...],-
          MYDRIVER.OPT/OPTIONS,-
          SYS$SYSTEM:SYS.STB/SELECTIVE_SEARCH
```

The linker will report that the image has no transfer address. You may ignore this message.

Once you have linked or relinked a driver, you should copy its image to the SYS$LOADABLE_IMAGES or SYS$SYSTEM directory. The SYSGEN LOAD and CONNECT commands first search for a driver in the SYS$LOADABLE_IMAGES directory. If they do not find the driver, they then search the SYS$SYSTEM directory.

E.3 Adapting Device Drivers to Run on a VMS Multiprocessing System

VMS now contains several new routines that enforce synchronization in a symmetric multiprocessing (SMP) environment. Drivers do not generally call these routines explicitly, but rather invoke VMS-supplied macros that synchronize as appropriate to the processing environment. There are only a few instances in which existing non-Digital-supplied drivers must change to conform to the new synchronization mechanisms. This section outlines those instances; Chapter 3 describes synchronization rules in greater detail.

E.3.1 Specifying the Fork Lock Index

To adapt a driver to execute properly in a VMS multiprocessor environment, you must replace all instances of UCB$B_FIPL (or FKB$B_FIPL) with UCB$B_FLCK (or FKB$B_FLCK). In addition, you must replace the invocation of the DPT_STORE macro that defined the driver's fork IPL with one that defines the driver's fork lock index, as follows:

Old: DPT_STORE UCB,UCB$B_FIPL,B,8

New: DPT_STORE UCB,UCBB_FLCK,B,SPLC_IOLOCK8

To use the new symbol, include the $UCBDEF structure definition macro in the driver. Fork lock (and other spin lock) indexes, such as SPL$C_IOLOCK8, are defined by the $SPLCODDEF definition macro as invoked by DPTAB. Replace fork IPLs with the corresponding fork lock index according to the following list:

IPL	Fork Lock Index
8	SPL$C_IOLOCK8
9	SPL$C_IOLOCK9
10	SPL$C_IOLOCK10
11	SPL$C_IOLOCK11

All structure definition macros can be found in SYS$LIBRARY:LIB.MLB.

Drivers rarely need to obtain a fork lock explicitly. VMS places the driver fork process into execution (originally by EXE$INSIOQ and, by implication, by IOC$REQCOM) at fork IPL holding the appropriate fork lock. In addition, the fork dispatcher obtains the fork lock associated with the driver fork process before it restores its context and resumes its execution.

Note that, if a driver fork process is not placed into execution according to one of these means, it must obtain the fork lock itself. (See the discussion in Section E.3.6.2.)

E.3.2 Synchronizing Access to the Device Database with the Interrupt Service Routine

The device database consists of device and adapter registers, plus driver-specific UCB fields that record the status of a device. As these locations are primarily accessed by the driver's interrupt service routine, the driver fork process must take special care to synchronize with the interrupt service routine whenever it accesses them.

E.3.2.1 Synchronizing at Device IPL

VMS prior to Version 5.0 used the DSBINT macro to synchronize with the interrupt service routine at device IPL (UCB$B_DIPL), as follows:

```
DSBINT UCB$B_DIPL(R5)        ;Raise IPL to device IPL
                             ;Save current IPL on stack
;Access device data
.
.
.
ENBINT                       ;Restore saved IPL
```

To run correctly, this code should be modified so that it obtains the appropriate device lock, as follows:

```
DEVICELOCK -                   ;Secure device lock
   LOCKADDR=UCB$L_DLCK(R5),-   ;(also raises IPL to device
                               ; IPL)
   SAVIPL=-(SP)                ;Save current IPL on stack
;Access device data
.
.
.
DEVICEUNLOCK -                 ;Release device lock
   LOCKADDR=UCB$L_DLCK(R5),-
   NEWIPL=(SP)+                ;Restore old IPL from stack
```

E.3.2.2 Raising IPL to IPL$_POWER

If the device driver start-I/O routine (or fork process) raises IPL to IPL 31 (IPL$_POWER) to check for the occurrence of a power failure and to access device registers, it must ensure that it has explicitly synchronized with the device's database at device IPL. This means that the routine must first obtain the appropriate device lock, using the DEVICELOCK macro.

Because versions of VMS prior to Version 5.0 allowed only one processor, even in a VAX multiprocessor system, to execute kernel-mode code, the following code in a driver's start-I/O routine provided adequate synchronization:

```
     DSBINT                          ;Raise IPL to 31
                                     ;Save current IPL on stack
     BBC     #UCB$V_POWER,-
               UCB$W_STS(R5),30$     ;If clear, no power failure
     ;Service power failure
   .
   .
   .
     ;Branch
30$: ;Start device
   .
   .
   .
     WFIKPCH
```

To run correctly, the preceding code should be replaced with the following:

```
     DEVICELOCK -                     ;Secure device lock
        LOCKADDR=UCB$L_DLCK(R5),-     ;(also raises IPL to
                                      ; device IPL)
        SAVIPL=-(SP)                 ;Save current IPL on stack
     SETIPL #IPL$_POWER,-            ;Raise IPL to 31
        ENVIRON=UNIPROCESSOR         ;Avoid assembly-time warning
     BBC     #UCB$V_POWER, -
               UCB$W_STS(R5),30$     ;If clear, no power failure
     ;Service power failure
   .
   .
   .
     DEVICEUNLOCK -                  ;Release device lock
        LOCKADDR=UCB$L_DLCK(R5),-
        NEWIPL=(SP)+                 ;Restore old IPL from stack
   .
   .
   .
     ;Branch
30$: ;Start device
   .
   .
   .
     WFIKPCH                         ;Wait for interrupt
```

Here, the DEVICELOCK macro achieves synchronized systemwide access to the device registers. The SETIPL macro then synchronizes the local processor against its own powerfail interrupt event. The code does not need to synchronize systemwide against powerfail events, because its interest is truly limited to the local processor.

Note that the WFIKPCH macro releases the last acquisition of the device lock by the executing processor, restoring the old IPL prior to returning control to the caller's caller.

Refer to Chapters 3 and 8 for additional information on the synchronization rules imposed on a driver's start-I/O routine.

E.3.2.3 Synchronization Within the Interrupt Service Routine

As soon as it obtains the device unit's UCB in R5, the driver's interrupt service routine must issue the DEVICELOCK macro to synchronize with other code threads (such as the start-I/O routine and the timeout handling routine) that may access the device database at device IPL holding the device lock. Because the interrupt service routine is automatically called at device IPL, the DEVICELOCK macro invocation should specify **condition=NOSETIPL**. To save time, the macro should also specify **preserve=NO** so that code to preserve R0 is not executed.

For example:

```
DEVICELOCK -                        ;Obtain device lock
   LOCKADDR=UCB$L_DLCK(R5),-
   CONDITION=NOSETIPL,-             ;Do not bother to set IPL
   PRESERVE=NO                      ;Do not save R0
```

Similarly, the interrupt service routine should release the device lock when it no longer needs to access the device database. Generally, this is immediately after the routine regains control from the driver fork process and before it restores the saved registers and issues an REI instruction, as follows:

```
DEVICEUNLOCK -                      ;Release device lock
   LOCKADDR=UCB$L_DLCK(R5),-
   PRESERVE=NO                      ;Do not save R0
POPR    #^M<R0,R1,R2,R3,R4,R5>      ;Restore registers
REI                                 ;Exit from interrupt
```

Refer to Chapters 3 and 9 for additional information on the synchronization rules imposed on a driver's interrupt service routine.

E.3.3 Controller and Unit Initialization Routines

As discussed in Section 11.1, a device driver's controller and unit initialization routines are called during driver loading and reloading and during system recovery from a power failure.

In a VMS symmetrical multiprocessing environment, any logic in a driver's controller initialization routine or unit initialization routine that takes special action to service a power failure must adhere to the following rules:

- It cannot acquire any spin locks. Controller and unit initialization routines are called at IPL 31 during power failure recovery to reinitialize I/O devices before the processors are allowed to proceed with execution at lower IPLs. Because processors may have been holding spin locks at the time of the power failure, they will not be able to release them until after they resume execution. As a result, spin locks are not available to controller and unit initialization routines.

- It cannot perform any operation that requires the intervention of other processors in the system.

E.3.3.1 Permanently Allocating Map Registers and Buffered Data Paths

Because the map registers and buffered data paths of a UNIBUS adapter are shared by the devices residing on the bus, they are synchronized at a single fork IPL and, in a VMS multiprocessing system, by a single fork lock.

Prior to VMS Version 5.0, a unit initialization routine that permanently allocated map registers or a buffered data path could do so at IPL$_POWER (its calling IPL). Now, however, the map register and data path allocation routines require that the appropriate fork lock be held at the time of their calling. As a result, a unit initialization routine that permanently allocates these resources must fork before calling the allocation routine. The VMS fork dispatcher ensures that, when execution of the routine resumes, it is executing at fork IPL holding the fork lock.

The consequences of forking in a unit initialization routine are discussed at length in Section 11.1.5. Refer to Sections 14.2.2.2 and 14.2.1.2 for additional information on permanently allocating map registers and buffered data paths, respectively.

E.3.4 Timeout Handling Routine

In a VMS multiprocessing environment, the software timeout interrupt service routine calls a driver's timeout handling routine at device IPL, holding both the appropriate fork lock and device lock.

Previous editions of this manual have suggested that a timeout handling routine can explicitly lower its IPL from device IPL to fork IPL using a SETIPL instruction. This action assumed that the thread of code that resulted in the call to the routine originated in a software interrupt granted at IPL 7 (IPL$_TIMERFORK).

In a VMS multiprocessing system, such a forced lowering of IPL would break synchronization. In addition, similar assumptions about the origin of the calling code thread cannot be guaranteed. Instead, those timeout handling routines that must lower IPL should issue the IOFORK macro to fork.

See Section 10.2 for additional information on the timeout handling routine.

E.3.5 General Methods for Synchronizing Kernel-Mode Code

In addition to the changes in the driver routines explicitly discussed in Sections E.3.2 and E.3.3, there may be other alterations required in device drivers and other kernel-mode code before they can execute successfully in a VMS symmetric multiprocessing environment. This section provides some general discussion of these changes. You can find additional information on multiprocessing synchronization in Chapter 3.

E.3.5.1 Using the Spin Lock Synchronization Macros

You must adapt most kernel-mode code that raises or lowers IPL so that it obtains appropriate synchronization in a VMS multiprocessing environment. Determine these locations by searching for instances of the system macros SETIPL, ENBINT, and DSBINT or for an instruction such as MTPR *x*, PR$_IPL (where *x* is an IPL value). Do not change those instances of the SETIPL and DSBINT macros intended to achieve synchronization only on the local processor. After careful inspection proves that the macro in question is intended to achieve local processor synchronization only, add the argument **environ=UNIPROCESSOR** to their invocations.

You should replace most instances of these macros with a LOCK, UNLOCK, FORKLOCK, FORKUNLOCK, DEVICELOCK, or DEVICEUNLOCK macro, as shown in Table E–3. You can substitute the appropriate usage of any of these macros wherever Table E–3 lists the LOCK and UNLOCK macros. The formats of the spin lock synchronization macros are fully described in *VMS Device Support Reference Manual*. Table 3–3 lists the system spin locks.

Table E–3 Converting IPL Synchronization to Spin Lock Synchronization

Existing Macro	Function	New Macro	Function
SETIPL **ipl** (where **ipl** is greater than 2)	Raise IPL	LOCK **lockname, lockipl**	Raise IPL, acquire spin lock
SETIPL **ipl** (where **ipl** is greater than 2)	Lower IPL from an IPL greater than 2	UNLOCK **lockname, lockipl**	Release spin lock, lower IPL
SETIPL **ipl** (where **ipl** is less than 3)	Lower IPL from an IPL less than 3	SETIPL **ipl**	Lower IPL
DSBINT **ipl**	Save current IPL and raise to specified IPL	LOCK **lockname, lockipl, saviplipl**	Save current IPL, raise IPL, acquire spin lock
ENBINT	Lower IPL and restore saved IPL	UNLOCK **lockname, lockipl**	Release spin lock, lower IPL

E.3.5.2 Interlocking Access to Data Cells and Queues

VMS now assigns spin lock protection to system resources as described in Table 3–3. The system time and timer queue are managed under the TIMER and HWCLK spin locks, as detailed in Section 3.1.3.

In a VMS multiprocessing environment, any thread of code that manipulates bit fields at different IPLs without spin lock protection must do so with interlocked instructions (for example, BBCCI and BBSSI).[1] Instances of the INSQUE and REMQUE instructions may need to be changed to use the INSQTI and REMQHI instructions, respectively, if they are issued to manipulate a queue at multiple IPLs. Certain cells, such as PCB$W_ASTCNT, must be incremented and decremented using an ADAWI instruction. INC*x* and DEC*x* instructions are not interlocked in a VMS multiprocessing system.

[1] It is illegal to intermix interlocked and noninterlocked instructions that refer to the same bit: for instance BBCC and BBCCI. Should any noninterlocked instruction refer to the same bit, the bit is not interlocked.

Spin locks explicitly protect various system queues and lists. For example, the AST queue in the process control block (PCB$L_ASTQFL) is synchronized by the SCHED spin lock and the variable region of nonpaged pool is protected by the POOL spin lock.

A fork lock implicitly protects the following adapter resource wait queues (at the specified listheads) at fork IPL, as long as the drivers for all devices on the adapter that require the resources use the same fork lock.

Listhead		
Since VMS V5.0	**Prior to V5.0**	**Name**
UCB$L_IOQFL	Same	Pending-I/O queue
ADP$L_DPQFL	Same	UNIBUS buffered data path wait queue
ADP$L_MRQFL	Same	UNIBUS/Q22 bus map register wait queue
ADP$L_MR2QFL	Same	Q22 bus alternate map register wait queue

Because a single spin lock cannot control access to list items that must be accessed by code threads executing at different IPLs, VMS now provides either a processor-specific queue or a self-relative queue for such items. The following queues (at the specified listheads) are now, processor-specific queues whose forward and backward links are contained in the per-CPU database (described in the *VMS Device Support Reference Manual* data structures).

Listhead		
Since VMS V5.0	**Prior to V5.0**	**Name**
CPU$Q_SWIQFL	SWI$GL_FQFL	Software interrupt queue listhead
IOC$GQ_POSTIQ	IOC$GL_PSFL	I/O postprocessing queue

The following queues are now self-relative queues whose forward and backward links are contained in the data area of the system loadable image SYSTEM_PRIMITIVES.EXE.[2] All system macros and routines that access these queues have been converted to access them with the INSQTI and REMQHI interlocked instructions.

Listhead		
Since VMS V5.0	**Prior to V5.0**	**Name**
IOC$GQ_SRPIQ	IOC$GL_SRPFL	Nonpaged pool SRP lookaside list
IOC$GQ_LRPIQ	IOC$GL_LRPFL	Nonpaged pool LRP lookaside list
IOC$GQ_IRPIQ	IOC$GL_IRPFL	Nonpaged pool IRP lookaside list

[2] System cell EXE$AR_SYSTEM_PRIMITIVES contains the address of this image; the macro $$SYSTEM_PRIM_DATADEF in SYS$SYSTEM:LIB.MLB defines offsets into its data area.

E.3.6 Miscellaneous Conversion Tasks

This section describes those activities performed by some kernel-mode code threads that should be examined in the course of converting them to run in a VMS multiprocessing environment.

E.3.6.1 Reading the System Time

As discussed in Section 3.1.3, because EXE$GQ_SYSTIME can only be changed or compared with multiple instructions, any code thread in a multiprocessing system that must obtain a consistent copy of the quadword must first acquire proper synchronization.

VMS now supplies the READ_SYSTIME macro to simplify this procedure. It has the following format, where **dst** is the quadword destination where the macro returns the system time:

```
READ_SYSTIME dst
```

Use of the READ_SYSTIME macro is subject to the following restrictions:

- IPL must be less than 23.
- The processor must be executing in kernel mode.
- When using the macro within pageable program sections executing at IPL 2 and below, you must ensure that the pages involved are locked in memory.

E.3.6.2 Calling the Driver Fork Process from a TQE

Whenever VMS places a driver fork process into execution, it ensures that it is synchronized with other processes at that fork level. In other words, if it is generated by the conclusion of I/O preprocessing (EXE$INSIOQ), the completion of a previous I/O request on a device unit (IOC$REQCOM), or the operation of the fork dispatcher, the driver fork process is placed into execution at the correct fork IPL, holding the corresponding fork lock.

As an example, consider a driver fork process activated by a timer wakeup associated with a timer queue element (TQE) previously queued by the driver. The software timer interrupt service routine does raise IPL to IPL 8 (IPL$_SYNCH) and obtain certain spin locks prior to dequeuing the TQE and placing it into execution, but it does *not* obtain the driver's fork lock. Thus, even though the driver's fork IPL may be IPL$_SYNCH, the driver will not be properly synchronized at fork level unless it first obtains the appropriate fork lock.

E.3.6.3 Invalidating Translation Buffer Entries

Prior to VMS Version 5.0, privileged code that changed a valid page-table entry (PTE) could flush the stale PTE from the processor's translation buffer by using the INVALID macro or writing directly to the Translation Buffer Invalidate Single (TBIS) processor register. Similarly, it could invalidate the entire translation buffer by using the INVALID macro or writing to the Translation Buffer Invalidate All (TBIA) processor register.

In a VMS symmetric multiprocessing environment, processors must not use previously buffered PTE contents while another processor is changing that PTE. Once the PTE has been changed, other processors must flush the stale translation buffer entry for the PTE. To accomplish this, VMS has replaced the INVALID macro with the INVALIDATE_TB macro.

The INVALIDATE_TB macro flushes a single PTE or all PTEs from the processor translation buffers in either a VAX uniprocessor or multiprocessor system. In updating privileged code written prior to VMS Version 5.0, you must replace any instances of an INVALID macro or of an MTPR instruction to PR$_TBIS or PR$_TBIA with a suitable invocation of the INVALIDATE_TB macro.

The *VMS Device Support Reference Manual* contains a description of the INVALIDATE_TB macro.

E.3.6.4 Unsupported Use of the IRP

The VMS multiprocessing code employs a portion of the IRP (the 24 bytes following IRP$L_KEYDESC), previously used only as a fork block by the VMS disk and tape class drivers, to effect the transfer of an I/O request from a processor with no access to the device to another processor that does have access.

A driver that uses this portion of the IRP to store data can lose this data when the VMS I/O initiation routine (IOC$INITIATE) attempts to transfer the request to the driver's start-I/O routine. VMS I/O initiation occurs when an FDT routine calls EXE$QIODRVPKT or when the driver issues the REQCOM macro to complete the current I/O request.

E.3.6.5 Poor Man's Lockdown

Certain privileged code, written prior to VMS Version 5.0, utilizes a technique, commonly known as "poor man's lockdown," whereby one or two pages of code are locked into a process or system working set as a side effect of elevating IPL. Such code has one of the following forms:

```
        ASSUME  10$-. LE 511      ; Check for contiguity of pages
        SETIPL   10$
        .
        .
        .
        ;Code to be locked into memory
        .
        .
        .
        RSB
10$:    .LONG  IPL$xxxx
```

The effect of this coding technique is that, because the system must determine the value of the argument to the SETIPL macro from location 10$, it must fault into memory the page in which 10$ resides. As a result, before the code actually elevates IPL, the pages in which the SETIPL macro and 10$ reside will become memory-resident. In this way, the code can avoid a page fault while executing the code between the SETIPL and 10$ at elevated IPL. The ASSUME macro guarantees that the pages to be faulted are contiguous.

This technique has several limitations:

- It cannot lock more than two virtually contiguous pages.
- Beginning with VMS Version 5.0, it is only useful in locking process pages, not system pages. In a VMS multiprocessing system, a page in the system working set could be faulted in by one processor, only to be removed from the system working set by another processor.

  To lock system pages, you must use the LOCK_SYSTEM_PAGES and UNLOCK_SYSTEM_PAGES macros as described in the *VMS Device Support Reference Manual*. (Note that you cannot use these macros to lock per-process pages in memory.)
- Prior to VMS Version 5.0, IPLs were the means by which system tasks were prioritized and access to system data was synchronized. Code executing at an elevated IPL would effectively block other code in the system from executing at or below that IPL. Now with symmetric multiprocessing, merely raising IPL does not synchronize systemwide activity nor enforce orderly access to data.

  Sometimes it may only be necessary to block tasks or synchronize activity on the local processor. In these instances, raising IPL provides sufficient synchronization and "poor man's lockdown" behaves as it did before VMS Version 5.0. For instance, use of "poor man's lockdown" to lock a code segment executing at IPL$_RESCHED effectively prevents process deletion and rescheduling while the code executes at a nonpageable IPL. However, if a locked code segment must access system data structures at an elevated IPL—for instance, at IPL$_SYNCH—it must obtain the spin lock associated with the database by using one of the spin lock synchronization macros (LOCK, FORKLOCK, or DEVICELOCK). After accessing the data, it must release the acquired spin lock by invoking UNLOCK, FORKUNLOCK, or DEVICEUNLOCK.

E.3.7 Troubleshooting a Device Driver in a Multiprocessing System

If the full-checking synchronization image has been loaded into memory, the spin lock acquisition and releasing routines perform certain activities that aid in the debugging and tuning of a VMS multiprocessing system. These activities include the following:

- Enforcement of the spin lock ranking and IPL requirements. The means by which the multiprocessing synchronization routines accomplish this are discussed in Section E.3.7.1.
- Recording, for each spin lock, the last eight PCs that acquired or released the spin lock. These PCs are located at offset SPL$L_OWN_PC_VEC in the spin lock data structure (SPL). You can use the SDA command SHOW SPINLOCKS/FULL to display the contents of the PC list.
- Tallying, for each spin lock, the number of successful acquisitions and the number of failed acquisitions in SPL$Q_ACQ_COUNT and SPL$L_BUSY_WAITS, respectively.

Section E.1.1 explains the settings of the MULTIPROCESSING system parameter that produce the full-checking synchronization environment.

The full-checking synchronization environment contains a mechanism for producing bugcheck messages that describe the detection of serious synchronization problems in the system. In most instances, these problems are caused by a non-Digital-supplied device driver that does not adhere to multiprocessing synchronization rules.

This section describes the bugchecks that are possible in a VMS full-checking synchronization environment and the SDA commands that aid in the investigation of a multiprocessing system failure. It concludes with a brief description of changes to the XDELTA debugger since VMS Version 5.0.

E.3.7.1 Multiprocessing Bugchecks

In order to obtain a spin lock or fork lock, a processor must be executing at an IPL no higher than the lock's synchronization IPL (SPL$B_IPL). Additionally, the processor cannot obtain a spin lock or fork lock if the lock's rank (SPL$B_RANK) is lower than that of any locks the processor currently holds. To release a spin lock, a processor must be executing at or above the IPL at which it originally acquired the lock. However, a processor can release spin locks in any order of rank. (See Table 3–3 for additional information on spin lock IPL and rank requirements.)

In a full-checking synchronization environment, violation of spin lock synchronization will produce the bugchecks described in Table E–4.

Table E–4 Bugchecks Produced Within Full-Checking Synchronization

SPLIPLHIGH	A processor has attempted to acquire a spin lock at an IPL higher than the IPL associated with spin lock synchronization (SPL$B_IPL). SMP$ACQUIRE (called by the LOCK and FORKLOCK macros with **condition=NOSETIPL** not specified) signals this bugcheck. A processor has attempted to acquire a device lock—not already owned by the acquiring processor—at an IPL higher than the IPL associated with device lock synchronization (SPL$B_IPL). SMP$ACQUIREL (called by the DEVICELOCK macro with **condition=NOSETIPL** *not* set) signals this bugcheck.
SPLIPLLOW	A processor has attempted to unconditionally or conditionally release a spin lock or device lock at an IPL lower than the IPL at which it originally acquired it. SMP$RELEASE and SMP$RESTORE (called by the UNLOCK and FORKUNLOCK macros) and SMP$RELEASEL or SMP$RESTOREL (called by the DEVICEUNLOCK macro) signal this bugcheck.
SPLACQERR	A processor has attempted to acquire a spin lock while holding a higher ranked spin lock. SMP$ACQUIRE, SMP$ACQUIREL, and SMP$ACQNOIPL (called by the LOCK, FORKLOCK, and DEVICELOCK macros) signal this bugcheck.
SPLRELERR	An attempt has been made to completely release a spin lock not owned by the releasing processor. SMP$RELEASE and SMP$RELEASEL (called by the UNLOCK, FORKUNLOCK, and DEVICEUNLOCK macros) signal this bugcheck.
SPLRSTERR	An attempt has been made to conditionally release a spin lock not owned by the releasing processor. SMP$RESTORE and SMP$RESTOREL (called by the UNLOCK, FORKUNLOCK, and DEVICEUNLOCK macros when **condition=RESTORE** is specified) signal this bugcheck.

E.3.7.2 Analyzing a Multiprocessing System Failure

When invoked to analyze either a crash dump or a running system, the VMS System Dump Analyzer (SDA) establishes a default context for itself from which it interprets certain commands.

When the subject of analysis is a VMS uniprocessing system, SDA's context is solely **process context**. That is, SDA can interpret its process-specific commands in the context of either the process current on the uniprocessor or some other process in some other scheduling state. When initially invoked to analyze a crash dump, SDA's process context defaults to the process that was current at the time of the crash. When invoked to analyze a running system, process context is initially that of the current process: that is, the one executing SDA. Change SDA's process context by entering commands in any of the following forms:

SET PROCESS/INDEX=**nn**
SET PROCESS **name**
SHOW PROCESS/INDEX=**nn**

When invoked to analyze a crash dump from a VMS multiprocessing system with more than one active CPU, SDA maintains a second dimension of context—its **CPU context**—that allows it to display certain processor-specific information, such as the reason for the bugcheck exception, the currently executing process, the current IPL, the contents of processor-specific registers, the interrupt stack pointer (ISP), and the spin locks owned by the processor. When invoked to analyze a multiprocessor's crash dump, the SDA CPU context defaults to that of the processor that induced the system failure.

Note: When you use SDA to analyze a running system, CPU context is not accessible to SDA. As a result, the SET CPU and SHOW CPU commands are not permitted.

Change the SDA CPU context by using any of the following commands:

SET CPU **cpu-id**
SHOW CPU **cpu-id**
SHOW CRASH

Changing CPU context involves an implicit change in process context in one of the following ways:

- If there is a current process on the CPU made current, SDA process context is changed to that of the CPU's current process.
- If there is no current process on the CPU made current, SDA process context is undefined and no process-specific information is available until SDA process context is set to that of a specific process.

Changing process context can involve a switch of CPU context as well. For instance, if you enter a SET PROCESS command for a process that is current on another CPU, SDA will automatically change its CPU context to that of the CPU on which the process is current. The following commands can have this effect if the **name** or index number (**nn**) refers to a current process:

SET PROCESS **name**

SET PROCESS/INDEX=**nn**
SHOW PROCESS **name**
SHOW PROCESS/INDEX=**nn**

E.3.7.2.1 Investigating the Status of Spin Locks

SDA now includes the command SHOW SPINLOCKS. The SHOW SPINLOCKS command displays various levels of information about system spin locks, fork locks, and device locks that help investigations of system failures caused by synchronization violations.

For each spin lock, fork lock, or device lock in the system, SHOW SPINLOCKS provides the following information:

- Name of the spin lock (or device name for the device lock)
- Address of the spin lock (SPL) structure
- The owner CPU's CPU ID
- IPL at which allocation of the lock is synchronized on a local processor
- Number of nested acquisitions of the spin lock by the processor (depth of ownership)
- Rank of the spin lock
- Number of processors waiting to obtain the spin lock
- Spin lock index

SHOW SPINLOCKS/BRIEF produces a condensed display of this same information.

If the VAX system under analysis had been executing with full-checking synchronization enabled (that is, with the MULTIPROCESSING system parameter set to 1 or 2), SHOW SPINLOCKS/FULL adds to the spin lock display the last eight PCs at which the lock was acquired or released.

E.3.7.3 Using XDELTA on SMP Systems

Only one processor in a VMS multiprocessing environment can be in XDELTA at a time. If one processor attempts to enter XDELTA while another processor is using XDELTA, it waits until the other processor has exited XDELTA. If the processor using XDELTA sets a breakpoint, other SMP processors are aware of the breakpoint. Therefore, when the code with the XDELTA breakpoint is executed on another processor, that processor will stop at the specified breakpoint and wait to enter XDELTA.

XDELTA uses its own system control block (SCB) to direct all interrupt handling to an error handling routine in XDELTA. Therefore, an error encountered by XDELTA will not affect any of the other processors which share the standard system SCB.

Glossary

ACF: See *configuration control block*.

ACP: See *ancillary control process*.

active set: In a VMS symmetric multiprocessing system, those processors that have been bootstrapped into the system, have undergone initialization, and are capable of scheduling and executing processes. Together, the primary processor and all secondary processors make up a system's active set. Compare with *available set*.

adapter control block (ADP): A structure in the I/O database that describes an I/O adapter (or VAXBI device) and its resources.

ADP: See *adapter control block*.

AEN: See *asynchronous event notification*.

affinity: In a VMS symmetric multiprocessing system, a close association of a device or a process with a specific processor or set of processors in the system. See *device affinity* and *process affinity*.

allocate a device: To reserve a particular device unit for exclusive use. A user process can allocate a device only when that device is not allocated by any other process.

ancillary control process (ACP): A process that acts as an interface between user software and an I/O driver. An ACP provides functions supplemental to those performed by the driver, such as file and directory management.

Examples of ACPs are the magnetic tape ACP (MTAACP) and the network ACP (NETACP).

ASMP: See *asymmetric multiprocessing*.

assign a channel: To establish the necessary software linkage between a user process and a device unit before a user process can communicate with that device. A user process requests the system to assign a channel and the system returns a channel number.

AST: See *asynchronous system trap*.

ASTLVL: See *asynchronous system trap level*.

asymmetric multiprocessing (ASMP): A multiprocessing configuration in which the processors are not equal in their ability to execute operating system code. In general, a single processor is designated as the primary, or master, processor; other processors are the slaves. The slave processors are limited to performing certain tasks, whereas the master processor can perform all system tasks. Contrast with *symmetric multiprocessing*.

asynchronous event notification (AEN): SCSI protocol allowing a SCSI device that is usually a target to inform the processor (usually the initiator) that an event has occurred asynchronously with respect to the processor's current stream of execution.

asynchronous system trap (AST): A software-simulated interrupt that passes control to a user-defined routine. ASTs enable a user process to be notified of the occurrence of a specific event asynchronously with respect to the execution of the user process.

If a user process has defined an AST routine for an event, the system interrupts the process and executes the AST routine when that event occurs. When the AST routine exits, the system resumes execution of the process at the point where it was interrupted.

asynchronous system trap level (ASTLVL): A value kept in an internal processor register that is the highest access mode for which an AST is pending. The AST does not occur until the current access mode drops in privilege (rises in numeric value) to a value greater than or equal to ASTLVL. Thus, an AST for an access mode will not be delivered while the processor is executing in a more privileged access mode.

attached processor: See *secondary processor*.

available set: In a VMS symmetric multiprocessing system, those processors that have passed the system's power-on hardware diagnostics and may or may not be actively involved in the system. The available set includes the active set. Compare with *active set*.

backplane interconnect: An internal processor bus that allows I/O device controllers to communicate with main memory and the central processor. These I/O controllers may reside on the same bus as memory and the central processor (for instance, in a VAX 82*x*0/83*x*0 system), or they may be on a separate bus entirely (for instance, in a VAX 8600/8650 system). In the latter case, an I/O adapter enables and controls the communications between the I/O bus and the processor and memory.

The backplane interconnect is called the synchronous backplane interconnect (SBI) in the VAX–11/780 and VAX 8600/8650 systems, the CPU-to-memory interconnect (CMI) in the VAX–11/750 system, the VAXBI in the VAX 82*x*0/83*x*0 systems, and the memory interconnect (NMI or XMI) in VAX 85*x*0/8700/88*x*0 and VAX 6000-series systems. The MicroVAX processors use the Q22 bus as a backplane.

base register: A general register that contains the base address (the address of the first entry) of a list, table, array, or other data structure.

buffered data path: A UNIBUS adapter data path that transfers several bytes of data in a single backplane-interconnect transfer.

buffered I/O: An I/O operation, such as terminal or mailbox I/O, in which an intermediate buffer from the system's buffer pool is used instead of a buffer in process space. See also *direct I/O*.

bugcheck: The operating system's diagnostic that detects and reports internal inconsistencies. If the system can continue running, it declares a **nonfatal bugcheck** and reports it in an error log entry. A serious error results in a **fatal bugcheck**. As a result of a fatal bugcheck, the system shuts itself down in an orderly fashion.

busy wait: See *spin wait*.

CALL instructions: The processor instructions CALLG (Call Procedure with General Argument List) and CALLS (Call Procedure with Stack Argument List).

capability: In a VMS symmetric multiprocessing environment, an attribute of a single processor or set of processors. The capabilities required by a given process determine the set of processors on which it can be scheduled. For instance, the VMS routine that maintains the system time can execute only on the processor that has the timekeeper capability.

CCB: See *channel control block*.

channel: A logical path connecting a user process to a physical device unit. A user process requests the operating system to assign a channel to a device so the process can communicate with that device. See also *controller data channel*.

channel control block (CCB): A structure in the I/O database maintained by the Assign-I/O-Channel system service to describe the device unit to which a channel is assigned.

channel request block (CRB): A structure in the I/O database that describes the activity on a particular controller. The CRB for a controller contains pointers to the queue of drivers waiting to access a device through the controller.

class driver: See *SCSI class driver*.

command descriptor block: Structure created by a SCSI class driver (or an application using the VMS generic SCSI class driver) in order to initiate a request of a device on the SCSI bus.

configuration control block (ACF): A structure in the I/O database used by the autoconfiguration facility of the System Generation Utility to describe the device it is adding to the system. The information stored in the ACF might be useful to a device driver's unit delivery routine.

configuration register: A control and status register for an I/O adapter (for example, a UNIBUS adapter). It resides in the adapter's I/O space.

connection: Logical link between a SCSI class driver and a device on the SCSI bus, involving the binding of the class driver to the VMS SCSI port driver. The connection allows the driver to issue commands to the SCSI device.

The class driver invokes the SPI$CONNECT macro to perform this linkage. A connection lasts throughout the run-time life of a system; a SCSI class driver should never need to break a connection.

connect-to-interrupt: A function by which a process connects to a device interrupt vector. To perform a connect-to-interrupt, the process must map to the physical pages in the I/O space that contain the vector.

console: The manual control unit integrated into the central processor. The console includes a serial-line interface connected to a hardcopy terminal. This enables the operator to start and stop the system, monitor system operation, and run diagnostic programs.

console terminal: The terminal connected to the central processor's console.

context: The environment of an activity. See also *process context, hardware context, and software context.*

control and status register (CSR): A control and status register for a device or controller. It resides in the processor's I/O space.

controller data channel: A logical path to which the driver of a device that shares a controller must gain access before it can use the controller to activate a device.

CRB: See *channel request block.*

CSR: See *control and status register.*

database: A collection of related data structures; all the occurrences of data described by a database management system.

data structure: Any table, list, array, queue, or tree whose format and access conventions are well defined for reference by one or more images.

DDB: See *device data block.*

DDT: See *driver dispatch table.*

device affinity: In a VMS symmetric multiprocessing system, a close association of a device with a specific processor or set of processors in the system. There are three dimensions to device affinity in a VMS system. First, physical connectivity describes those devices that are directly accessible only to the primary processor or to all processors. Secondly, affinity is a software mechanism that defines those processors that can initiate an I/O operation on the device. Finally, interruptibility describes the set of processors that can receive interrupts from a device.

device data block (DDB): A structure in the I/O database that identifies the generic device/controller name and driver name for a set of devices that share the same controller.

device driver: The set of instructions and tables that handles physical I/O operations to a device.

device ID: See *SCSI device ID*.

device interrupt: An interrupt received on interrupt priority levels 20 through 23. Device interrupts can be requested only by devices, controllers, and memories.

device lock: In a VMS symmetric multiprocessing system, a dynamic spin lock the ownership of which synchronizes device-specific code that executes at device IPL. A device lock is associated with each adapter or controller in the system. See *spin lock*.

device register: A location in controller logic used to request device functions (such as I/O transfers) and/or report status.

device unit: One device and its controlling logic (for example, a disk drive or terminal). Some controllers can have several device units connected to a single controller (for example, mass-storage controllers).

diagnostic program: A program that tests hardware, firmware, peripherals logic, or memory, and that reports any faults it detects.

direct data path: A UNIBUS adapter data path that transfers several bytes of data in a single backplane-interconnect transfer.

direct I/O: An I/O operation in which VMS locks the pages containing the associated buffer in physical memory for the duration of the I/O operation. The I/O transfer takes place directly from the process's buffer. Contrast with *system buffered I/O*.

direct-memory-access (DMA) transfer: The type of I/O transfer by which a device controller accesses memory directly and, as a result, can transfer a large amount of data without requesting a processor interrupt after each of the smaller amounts. Contrast with *programmed-I/O (PIO) transfer*.

DPT: See *driver prologue table*.

drive: The electromechanical unit of a mass storage device on which a recording medium (disk cartridge, disk pack, or magnetic tape reel) is mounted.

driver dispatch table (DDT): A table in a driver that lists the addresses of the entry points of standard driver routines and the sizes of diagnostic and error message buffers for the device.

driver prologue table (DPT): A table in a driver that describes the driver and the type of device it controls to the VMS procedure that loads drivers into the system.

dynamic load balancing: A method of work distribution in which the operating system ensures that the system work load is evenly distributed among the processors. Dynamic load balancing in a VMS symmetric multiprocessing system is a direct effect of the implementation of the scheduler. In a VMS multiprocessing system, processors independently and continually look for processes to execute from a common pool of such processes.

ECC: Error-Correction Code.

error logger: A system process that empties the error logging buffers and writes the error messages into the error file. Errors logged by the system include memory errors, device errors and timeouts, and interrupts with invalid vector addresses.

exception: An event detected by the hardware or software (other than an interrupt or jump, branch, case, or call instruction) that changes the normal flow of instruction execution.

An exception is always caused by the execution of an instruction or set of instructions (whereas an interrupt is caused by an activity in the system that is independent of the current instruction).

There are three types of hardware exceptions: traps, faults, and aborts. Examples are attempts to execute a privileged or reserved instruction, trace traps, page faults, compatibility-mode faults, execution of breakpoint instructions, and arithmetic traps.

executive: The software that provides the basic control and monitoring functions of the operating system.

extended QIO processor: The facility that supplements the QIO driver's functions when the driver performs virtual I/O operations on file-structured devices (Files–11 On-Disk Structure Level 2). The XQP executes as a kernel-mode thread in the process of its caller.

FDT: See *function decision table.*

FDT routines: Driver routines called by the $QIO system service to perform device-dependent preprocessing of an I/O request.

fork block: That portion of a data structure, such as the unit control block, which contains a driver's context while the driver is waiting for an event or a resource. A driver awaiting the processor resource has its UCB fork block linked into a processor-specific fork queue.

fork dispatcher: A VMS interrupt service routine that is activated by a software interrupt on the local processor at a fork IPL. Once activated, it obtains the fork lock associated with the fork IPL and dispatches driver fork processes from a fork queue until no processes remain in the queue for that IPL.

fork lock: In a VMS symmetric multiprocessing system, a static spin lock the ownership of which synchronizes the right of a driver's fork process to execute at its associated fork IPL. See *spin lock.*

fork process: A process with a minimal context that executes instructions under a set of constraints: it executes at raised interrupt priority levels; it uses R0 through R5 only (other registers must be saved and restored); it executes in the system's virtual address space; it can refer to and modify static storage that is never modified by procedures that execute at a higher IPL. VMS uses software interrupts, spin locks, fork processes, and resource wait queues to synchronize executive operations.

fork queue: A processor-specific queue of fork blocks that are awaiting activation at a particular IPL by the VMS fork dispatcher.

function code: See *I/O function code*.

function decision table (FDT): A table in the driver that lists all valid function codes for the device and lists the addresses of preprocessing routines associated with each valid function of the device.

function modifier: See *I/O function modifier*.

generic device name: A device name that identifies the type of device but not a particular unit; a device name in which the specific controller and/or unit number is omitted (for example, MB).

hardware context: The values contained in the following registers while a process is executing:

- The PC
- The PSL
- The 14 general registers (R0 through R13)
- The four processor registers (P0BR, P0LR, P1BR, and P1LR) that describe the process's virtual address space
- The SP for the access mode in which the processor is executing
- The contents to be loaded in the SP for every access mode other than the current access mode

When a process is executing, its hardware context is continually being updated by the processor. When a process is not executing, its hardware context is stored in its hardware PCB.

hardware process control block (hardware PCB): A data structure known to the processor that contains the hardware context when a process is not executing. A process's hardware PCB resides in its process header (PHD).

IDB: See *interrupt dispatch block*.

Initiator: A SCSI device (usually the host processor) that requests another SCSI device (the target) to perform an operation.

interrupt: An event other than an exception or a branch, jump, case, or call instruction that changes the normal flow of instruction execution. Interrupts are generally external to the process executing when the interrupt occurs. See also *device interrupt*, *software interrupt*, and *urgent interrupt*.

interrupt dispatch block (IDB): A structure in the I/O database that describes the characteristics of a particular controller and points to devices attached to that controller.

interrupt priority level (IPL): The level at which a software or hardware interrupt is generated. There are 32 interrupt priority levels: IPL 0 is lowest, 31 is highest. The levels arbitrate contention for processor service. For example, a device cannot interrupt a processor if the processor is currently executing at an IPL greater than the IPL of the device's interrupt request.

interrupt service routine (ISR): A routine executed when a device interrupt occurs.

interrupt stack (IS): The processor-specific stack used when the processor is executing instructions in interrupt context. In the VMS operating system, all hardware interrupts (and all software interrupts above IPL 3) are serviced on a processor-specific interrupt stack and not one of the process stacks.

interrupt stack pointer (ISP): The pointer to the top of the interrupt stack.

interrupt vector: See *vector*.

I/O database: A collection of data structures that describe I/O requests, controllers, device units, volumes, and device drivers in a VMS system. Examples are the driver dispatch table, driver prologue table, device data table, unit control block, channel request block, I/O request packet, and interrupt dispatch block.

I/O driver: See *driver*.

I/O function: An I/O operation interpreted by the operating system and typically resulting in one or more physical I/O operations.

I/O function code: A 6-bit value specified in a $QIO system service call that describes the particular I/O operation to be performed (such as, read, write, rewind).

I/O function modifier: A 10-bit value specified in a $QIO system service call that modifies an I/O function code (for example: read terminal input, no echo).

I/O lockdown: The state of a page such that it cannot be paged or swapped out of memory.

I/O request packet (IRP): A structure in the I/O database that describes an individual I/O request. The $QIO system service creates an IRP for each I/O request. VMS and the driver of the target device use information in the IRP to process the request.

I/O rundown: An operating system function in which the system cleans up any I/O in progress when an image exits.

I/O space: The regions of physical address space that contain the configuration registers and device control and status register and data registers. These regions are physically noncontiguous.

I/O status block (IOSB): A data structure associated with the $QIO system service. This service optionally returns a status code, number of bytes transferred, and device/function-dependent information in an I/O status block. The information is not returned from the system service call, but filled in by VMS when the I/O request completes.

IPL: See *interrupt priority level*.

IRP: See *I/O request packet*.

ISP: See *interrupt stack pointer*.

ISR: See *interrupt service routine*.

limit: The size or number of items requiring system resources (such as mailboxes, locked pages, I/O requests, or open files) that a job is allowed to have at any one time during execution, as specified by the system manager in the user authorization file. See *quota*.

load balancing: A function of the operating system by which work is distributed equally among all processors in a system. For more information, see *static load balancing* and *dynamic load balancing*.

locking a page in memory: Making a page ineligible for either paging or swapping. A page stays locked in physical memory until VMS specifically unlocks it.

logical-I/O function: A set of I/O operations (for example, read-logical-block and write-logical-block) that allow restricted direct access to device-level I/O operations using logical block numbers.

logical unit number (LUN): Unique value, from 0 to 7, that identifies a physical or virtual device accessible by means of a SCSI device with respect to that device's SCSI device ID.

loosely coupled system: A multiprocessing system configuration consisting of separate operating systems that communicate through some message transfer mechanism. Contrast with *tightly coupled system*.

LUN: See *Logical unit number*.

machine check: An exception that is reported when the processor or an external adapter detects an internal error. If the machine check is recoverable, the machine check handler logs the condition in an error log entry. If an unrecoverable machine check occurs while the processor is in supervisor or user mode, the machine check handler reports the exception to that mode. However, if an unrecoverable machine check occurs in kernel or executive mode, a fatal bugcheck results. See also *exception* and *bugcheck*.

mailbox: A software data structure that is treated as a record-oriented device for interprocess communication (for example, the error logger and OPCOM read from systemwide mailboxes). Communication using a mailbox is similar to other forms of device-independent I/O. Senders write to a mailbox; the receiver reads from that mailbox.

map register: See *scatter-gather map*.

MASSBUS adapter (MBA): An interface device between the backplane interconnect and the MASSBUS.

memory interconnect: The name of the internal processor bus for the VAX–11/750 (CMI), VAX 85*x*0/8700/88*x*0 (NMI), VAX 6000 series (XMI), and VAX 9000 series (SCU).

multiprocessing system: A system containing two or more general purpose processors. These processors are connected through hardware so that they can work on the same application concurrently. See *asymmetric multiprocessing* and *symmetric multiprocessing*.

multiprogramming: A mode of operation in which hardware resources are shared among multiple, independent software processes.

nexus: A physical connection to the synchronous backplane interconnect (SBI). For example, when connected to the SBI, the central processor, memory subsystem, and I/O controllers are known as nexuses. See also *synchronous backplane interconnect*.

node: A VAXBI interface—such as a central processor, controller, or memory subsystem—that occupies one of 16 logical locations on a VAXBI bus. See also *VAXBI*.

offset: A displacement from the beginning of a data structure to the beginning of a field within that data structure. Offsets for items within a data structure usually have an associated symbol. The name of the symbol is used to refer to the field; its value is the offset.

page-frame number (PFN): The high-order 21 bits of the physical address of a page in physical memory.

page-table entry (PTE): The data structure that identifies the physical location and status of a page of virtual address space. When a virtual page is in memory, the PTE contains the page-frame number needed to map the virtual page to a physical page. When it is not in memory, the page-table entry contains the information needed to locate the page on secondary storage (disk).

parallel processing: A method of computing that occurs when a section of an application is divided into multiple tasks, and those multiple tasks are executed simultaneously on multiple processors.

PCB: See *process control block*.

PFN: See *page-frame number*.

physical address: The address used by hardware to identify a location in physical memory or on directly-addressable secondary storage devices such as disks. A physical-memory address consists of a page-frame number and the number of a byte within the page. A physical-disk-block address consists of a cylinder or track and a sector number.

physical address space: The set of all possible physical addresses that can be used to refer to locations in memory (memory space) or device registers (I/O space).

physical-I/O functions: A set of I/O functions that allows access to all device-level I/O operations except maintenance-mode operations.

PID: See *process identification*.

port: See *SCSI port*.

port driver: See *SCSI port driver*.

port ID: See *SCSI port ID*.

primary processor: The processor in a VMS symmetric multiprocessing system that is either logically or physically attached to the console device. Only the primary processor performs the initialization activities that define the VMS environment and prepare memory for the entire system. In addition, the primary processor serves as the system timekeeper.

process: The basic entity, scheduled by the system software, that provides the context in which an image executes. A process consists of an address space, hardware context, and software context.

process affinity: In a VMS symmetric multiprocessing system, a close association of a process with a specific processor or set of processors in the system. Process affinity can be indicated as either a requirement that a process run only on the processor with a specific CPU ID or on a processor or set of processors that have a needed capability. See *capability*.

process context: The hardware and software contexts of a process.

process control block (PCB): A data structure used to contain process context. The hardware PCB contains the hardware context. The software PCB contains the software context, which includes a pointer to the hardware PCB.

process identification (PID): A 32-bit value that uniquely identifies a process. Each process has a PID and a name.

process I/O channel: See *channel*.

process page tables: The page tables used to describe process virtual memory.

process priority: The priority assigned to a process for scheduling purposes. The operating system recognizes 32 levels of process priority, where 0 is lowest and 31 is highest. Levels 16 through 31 are used for real-time processes. The system does not modify the priority of a real-time process (although the system manager or the process itself might). Levels 0 through 15 are used for normal processes. The system can temporarily increase the priority of a normal process based on the activity of the process.

Contrast with *interrupt priority level*.

program section (psect): A portion of a program with a given protection and set of storage-management attributes. Program sections that have the same attributes are gathered together by the linker to form an image section.

programmed-I/O (PIO) transfer: The type of I/O transfer, largely conducted by the driver program, that requires processor intervention after each byte or word is transferred. Drivers for relatively slow devices, such as printers, card readers, terminals, and some disk and tape drives use PIO data transfers. Contrast with *direct-memory-access (DMA) transfer*.

PTE: See *page-table entry*.

Q22 bus: The hardware interconnect by which MicroVAX peripheral devices communicate with main memory and the processor.

QIO: Queue I/O Request system service. The VMS system service that services $QIO and $QIOW requests. The Queue I/O Request system service prepares an I/O request for processing by the driver and performs device-independent preprocessing of the request. This system service also calls driver FDT routines. See also *FDT routines*.

quota: The total amount of a system resource, such as CPU time, that a job is allowed to use in an accounting period, as specified by the system manager in the user-authorization file. See *limit*.

return status code: See *status code*.

SBI: See *synchronous backplane interconnect*.

scatter-gather map: A technique by which a set of physically discontiguous pages are made to seem contiguous to an I/O controller performing a direct-memory-access transfer. It is I/O adapter hardware that generally provides this means of mapping physical pages to I/O adapter address space.

SCDRP: See *SCSI class driver request packet*.

SCDT: See *SCSI connection descriptor table*.

SCSI: Refers to the American National Standard for Information Systems Small Computer System Interface–1 (X3.131–1986) or the ANSI Small Computer System Interface–2 (X3.131–1989). This standard defines mechanical, electrical, and functional requirements for attaching small computers to each other and to intelligent peripheral devices.

SCSI class driver: Component of the VMS SCSI class/port architecture that acts as an interface between the user and the SCSI port, translating I/O functions as specified in a user's $QIO request to a SCSI command targeted to a device on the SCSI bus. Although the class driver knows about SCSI command descriptor buffers, status codes, and data, it has no knowledge of underlying bus protocols or hardware, command transmission, bus phases, timing, or messages. A single class driver can run on any given MicroVAX/VAXstation system, in conjunction with the SCSI port driver that supports that system.

SCSI class driver request packet (SCDRP): A VMS data structure that contains information specific to an I/O request that a SCSI class driver must deliver to the port driver, such as the address of the SCSI command descriptor buffer. The class driver allocates the SCDRP, and places in it data it originally received in the I/O request packet (IRP), such as the $QIO system service parameters, I/O function, and the length and location of any user-specified buffer involved in a transfer.

SCSI connection descriptor table (SCDT): A VMS data structure that contains information describing a connection established between a SCSI class driver and the port, such as phase records, timeout values, and error counters. The SCSI port driver creates an SCDT each time a SCSI class driver, by invoking the SPI$CONNECT macro, connects to a device on the SCSI bus. The class driver stores the address of the SCDT in the SCSI device's UCB.

SCSI device ID: Unique value, from 0 to 7, representing a device on a specific SCSI bus. A VMS SCSI device ID corresponds to the line on the SCSI data bus on which a given device asserts itself and thus is an analog for the term SCSI ID.

Typically, a MicroVAX/VAXstation 3100 system processor is assigned device ID 6 and asserts itself at DB(6); a VAXstation 3520/3540 system processor is assigned device ID 7, and asserts itself at DB(7).

SCSI ID: See *SCSI device ID*.

SCSI port: The SCSI controller channel that controls communications to and from a specific SCSI bus in the system.

SCSI port descriptor table (SPDT): A VMS data structure that contains information specific to a SCSI port, such as the port driver connection database. The SPDT also includes a set of vectors, corresponding to the SPI macros invoked by SCSI class drivers, that point to service routines within the port driver. The SCSI port driver's unit initialization routine creates an SPDT for each SCSI port defined for a specific MicroVAX/VAXstation system and initializes each SPI vector.

SCSI port driver: Component of the VMS SCSI class/port architecture that transmits and receives SCSI commands and data. It knows the details of transmitting data from the local processor's SCSI port hardware across the SCSI bus. Although it understands SCSI bus phases, protocol, and timing, the SCSI port driver has no knowledge of which SCSI commands a given device supports, what status messages it returns, or the format of the packets in which this information is delivered. Strictly speaking, the port driver is a communications path. When directed by a SCSI class driver, the port driver forwards commands and data from the class driver onto the SCSI bus to the device. On any given MicroVAX/VAXstation system, a single SCSI port driver handles bus-level communications for all SCSI class drivers that may exist on the system.

SCSI port ID: A unique representation of a SCSI port (see *SCSI port*) identifying the SCSI bus it controls. Current legal port IDs are *A* and *B*, corresponding to a VMS controller ID.

secondary processor: The processor or processors in a VMS symmetric multiprocessing system that do not have the initialization and timekeeper responsibilities of the primary processor.

Small Computer System Interface: See *SCSI*.

small process: A system process that has no control region in its virtual address space and has an abbreviated context. Examples are the swapper and the null process. A small process is scheduled in the same manner as user processes, but must remain resident until it completes execution; it cannot be swapped.

shared memory: A generic term referring to any memory that can be accessed by two or more concurrent processes. In a VMS symmetric multiprocessing system, a single copy of the VMS operating system resides in memory. Each processor in the system can access this memory, as can any process executing on any processor.

SMP: See *symmetric multiprocessing*.

software context: The context maintained by VMS to describe a process. See also *software process control block (PCB).*

software process control block (software PCB): The data structure used to contain a process's software context. The operating system defines a software PCB for every process when the process is created.

The software PCB includes the following kinds of information about the process: current state; storage address, if the process is swapped out of memory; unique identification of the process; and address of the process header (which contains the hardware PCB). The software PCB resides in the system region of virtual address space. It is not swapped with a process.

start-I/O routine: The routine in a device driver that is responsible for obtaining needed resources and for activating the device unit. An example of a needed resource is the controller's data channel.

spin lock: In a VMS symmetric multiprocessing system, a semaphore associated with a set of system structures, fields, or registers whose integrity is critical to the performance of a specific operating system task. There are two types of spin lock. Static spin locks are assembled permanently into the system; the same static spin locks exist in the same memory locations in all VMS multiprocessing systems. A fork lock is a form of static spin lock. Dynamic spin locks are created as required by the I/O configuration of a system; as a result, the set of dynamic spin locks differs from processor to processor. A device lock is a form of dynamic spin lock. See *fork lock* and *device lock.*

spin wait: In a VMS symmetric multiprocessing system, an execution loop performed by a processor attempting to acquire a spin lock already owned by another processor in the system. This activity is also known as a busy wait.

SPDT: See *SCSI port descriptor table.*

static load balancing: A method of work distribution in which every process in an application is preassigned to a processor during process creation.

status code: A longword value that indicates the success or failure of a specific function. For example, system services always return a status code in R0 upon completion.

SVA: See *system virtual address.*

symmetric multiprocessing (SMP): A multiprocessing system configuration in which all processors have equal access to operating system code residing in shared memory and can perform all, or almost all, system tasks.

synchronous backplane interconnect (SBI): The part of the VAX–11/780, VAX–11 /785, and VAX 8600/8650 hardware that interconnects the processor, memory controllers, MASSBUS adapters, and the UNIBUS adapter.

system control unit (SCU): Functions as a memory interconnect and internal processor bus for the VAX 9000-series systems. Also connects to the I/O control unit and XJA adapters translating I/O addresses between the primary I/O buses (XMIs) and memory or processors in the system.

system page table (SPT): The data structure that maps the system virtual addresses, including the addresses used to refer to the process page tables. The SPT contains one PTE for each page of system virtual memory. The physical base address of the SPT is contained in a processor register called the System Base Register (SBR).

system virtual address (SVA): A virtual address identifying a location mapped to an address in system space.

target: A SCSI device that performs an operation requested by an initiator.

tightly coupled system: A multiprocessing system configuration consisting of multiple processors sharing a single copy of the operating system. These processors are connected so that they can communicate and share data. Contrast with *loosely coupled system*.

timeout: The expiration of the time limit in which a device is to complete an I/O transfer. The driver's wait-for-interrupt request specifies the timeout limit.

timer: A system process that maintains the time of day and the date. It is also alert for device timeouts and performs time-dependent scheduling upon request. The timer's interrupt service routine creates the timer process.

UCB: See *unit control block*.

UNIBUS adapter: An interface device between the backplane interconnect and the UNIBUS. In a VAX–11/780, VAX–11/785, or VAX 8600/8650 system this device is called the UBA. In a VAX–11/750 system, it is called the UBI. In a VAX 82*x*0 /83*x*0 or VAX 85*x*0/8700/88*x*0 system, it is called a DWBUA. In a VAX 6000-series system, it is called a DWMUA.

unit control block (UCB): A structure in the I/O database that describes the characteristics of a device unit and current activity on it. The unit control block also holds the fork block for its unit's device driver; the fork block is part of the UCB and is a critical part of a driver fork process. The UCB also provides a static storage area for the driver.

unit initialization routine: The routine that readies controllers and device units for operation. Controllers and device units require initialization after a power failure and during execution of the driver-loading procedure.

urgent interrupt: An interrupt received on interrupt priority levels 24 through 31. These can be generated only by the processor for the interval clock, serious errors, and power failures.

VAXBI: The part of the VAX 82*x*0/83*x*0 hardware that connects I/O adapters with memory controllers and the processor. In a VAX 85*x*0/8700/88*x*0 system, the part of the hardware that connects I/O adapters with the bus that interfaces with the processor and memory.

vector: A one-dimensional array.

An interrupt or exception vector is a storage location known to the system that contains the starting address of a routine to be executed when a given interrupt or exception occurs. The system defines separate vectors for each interrupting adapter and for classes of exceptions. Each system vector is a longword.

For the purpose of handling exceptions, users can declare up to two software-exception vectors (primary and secondary) for each of the four processor-access modes. Each vector contains the address of a condition handler, and is a longword.

virtual-I/O functions: A set of I/O functions that must be interpreted by an ancillary control process.

wait-for-interrupt request: A request made by a driver's start-I/O routine after it activates a device. The request causes the driver's fork process to be suspended until the device requests an interrupt or the device times out.

XDELTA: A software tool for debugging the VMS operating system and device drivers.

XQP: See *extended QIO processor*.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-343-4040 before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-DEC-DEMO (800-332-3366) using a 1200- or 2400-baud modem. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	————	Local Digital subsidiary or approved distributor
Internal[1]	————	USASSB Order Processing - WMO/E15 *or* U.S. Area Software Supply Business Digital Equipment Corporation Westminster, Massachusetts 01473

[1]For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

Reader's Comments

VMS Device Support Manual
AA–PBPWA–TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less _____

_____

_____

What I like best about this manual is _____

_____

_____

What I like least about this manual is _____

_____

_____

I found the following errors in this manual:

Page	Description
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

_____

_____

_____

_____

I am using **Version** _____ of the software this manual describes.

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Phone _____

--- Do Not Tear - Fold Here and Tape ---

digital™

No Postage
Necessary
if Mailed
in the
United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01–3/J35 110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

--- Do Not Tear - Fold Here ---

Cut Along Dotted Line

Reader's Comments

VMS Device Support Manual
AA–PBPWA–TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less _____

_____

_____

What I like best about this manual is _____

_____

_____

What I like least about this manual is _____

_____

_____

I found the following errors in this manual:

Page	Description
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

_____

_____

_____

_____

I am using **Version** _____ of the software this manual describes.

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Phone _____

--- Do Not Tear - Fold Here and Tape ---

digital™

No Postage
Necessary
if Mailed
in the
United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01–3/J35 110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

--- Do Not Tear - Fold Here ---

Cut Along Dotted Line